

## 版权注意事项：

- 1、书籍版权归作者和出版社所有
- 2、本PDF仅限用于个人获取知识，进行私底下的知识交流
- 3、PDF获得者不得在互联网上以任何目的进行传播
- 4、如觉得书籍内容很赞，请购买正版实体书，支持作者
- 5、请于下载PDF后24小时内删除本PDF。



非卖品！！严禁（售卖和上传互联网平台）！！  
仅供对书籍质量进行鉴定甄别！为是否购买正版实体书提供依据！！

# 微服务

## 设计原理与架构

MICROSERVICES  
Design Principle and Architecture

郑天民 / 著

深入剖析微服务架构设计理念与技术体系  
全面阐述实施微服务架构的系统方法与工程实践  
涵盖向微服务架构转型的思路、过程和案例



中国工信出版集团



人民邮电出版社  
POSTS & TELECOM PRESS



非卖品！！严禁（售卖和上传互联网平台）！！  
仅供对书籍质量进行鉴定甄别！为是否购买正版实体书提供依据！！

# 微服务

## 设计原理与架构

MICROSERVICES

Design Principle and Architecture

郑天民 / 著

人民邮电出版社  
北京



非卖品!! 严禁(售卖和上传互联网平台)!!  
仅供对书籍质量进行鉴定甄别! 为是否购买正版实体书提供依据!!

## 图书在版编目(CIP)数据

微服务设计原理与架构 / 郑天民著. — 北京: 人民邮电出版社, 2018.5  
ISBN 978-7-115-47882-5

I. ①微… II. ①郑… III. ①互联网络—网络服务器—程序设计 IV. ①TP368.5

中国版本图书馆CIP数据核字(2018)第025531号

## 内 容 提 要

本书内容主要包含实施微服务架构的一些方法论和工程实践, 首先, 通过对微服务架构的基本概念、服务建模、服务拆分和集成的介绍, 帮助读者全面理解微服务架构中的设计理念, 然后从微服务架构的基础组件、关键要素、实现框架以及管理体系等维度出发, 阐述实现微服务架构的工具和实践。最后, 本书还给出了从现有系统向微服务架构转型的思路、过程和案例分析。

本书面向立志于成为微服务架构师的后端服务开发人员, 读者不需要有很深的技术水平, 也不限于特定的开发语言; 不过, 熟悉 Java EE 常见技术并掌握一定系统设计基本概念, 有助于更好地理解书中的内容。同时, 本书也可以供具备不同技术体系的架构师同行参考阅读, 希望能给日常研发和管理工作带来启发和帮助。

著: 郑天民

- 
- ◆ 著 郑天民  
责任编辑 刘 博  
责任印制 沈 蓉 彭志环
  - ◆ 人民邮电出版社出版发行 北京市丰台区成寿寺路11号  
邮编 100164 电子邮件 315@ptpress.com.cn  
网址 <http://www.ptpress.com.cn>  
固安县铭成印刷有限公司印刷
  - ◆ 开本: 800×1000 1/16  
印张: 18.75 2018年5月第1版  
字数: 398千字 2018年5月河北第1次印刷
- 

定价: 59.80 元

读者服务热线: (010) 81055256 印装质量热线: (010) 81055316

反盗版热线: (010) 81055315



# 前言

随着互联网行业的飞速发展，快速的业务更新和产品迭代给系统开发过程和模式带来了新的挑战。业务需求层出不穷且变化不断、技术发展日新月异、团队规模从无到有快速扩张，因此，系统的复杂性以及对行业变化的快速应变能力等成为软件开发的核心问题。围绕这些问题，如何更为合理地划分系统和团队边界、如何更加有效地组织系统开发过程、如何通过技术手段识别和消除开发过程中的浪费，成为广大软件开发和技术管理人员所需要思考的问题。在这一时代背景下，微服务架构的出现为我们提供了一种具体的解决方案。

本书从微服务的基本概念出发，阐述了微服务架构的方方面面。除了具体实现工具和框架之外，本书还介绍了微服务架构的基本原理和技术体系，并阐述现有系统向微服务架构转型的系统方法，旨在为广大开发人员提供一套系统的、全面的微服务实施指南。

构建微服务架构是一项系统工程，涉及服务建模、实现技术、基础设施和研发过程等各个维度。本书从建模、实现和转型这三个特定角度出发，结合作者自身在互联网行业多年的技术应用与管理工作经验展开论述，介绍微服务架构设计相关的方法论和工程实践，具有较强的针对性和适用性。微服务架构是一个非常广泛的概念，本书整体上是“原理”结合“技术”的行文思路，不仅仅介绍微服务架构实现上的具体工具，更是对这些工具背后的原理和设计思想进行剖析，具备一定广度的同时也提供了对应深度的知识体系。

本书共分为四篇，共计八章内容，分别从不同的领域对微服务架构的各个方面展开讨论。

1. 直面微服务篇。从微服务的基本概念出发，阐述微服务架构的发展阶段、所具备的优势以及所面临的挑战，并给出实施微服务架构的系统方法。

2. 服务建模篇。关注于微服务建模，首先介绍服务建模方法，用于明确服务模型的各个维度和表现形式；然后对服务拆分和集成方法进行展开，侧重于从服务的依赖关系、数据、事务边界等维度出发讨论实现策略。

3. 服务实现篇。作为微服务架构实现过程中的主体知识部分，本篇从微服务架构基础组件、关键要素、实现技术和管理体系等四个角度切入，全面介绍微服务架构实现上的工具框架、技术原理和最佳实践。

4. 服务转型篇。从实际应用角度出发探讨如何在现有系统的基础上向微服务架构转型，一方面提供技术架构调整的方法和模式，另一方面也阐述了如何从组织过程管理角度出发向微服务架构转型。

通过对本书的系统学习，读者将对微服务架构的基本原理、设计思想和实现方式有全面而深入的了解，为后续的工作和学习铺平道路。



图书在版编目(CIP)数据

本书的撰写成功要感谢我的家人，特别是我的妻子章兰婷女士，在我占用大量晚上和周末时间的情况下，能够给予极大的支持和理解。感谢以往以及现在公司的同事们，身处业界领先的公司和团队让我得到很多学习和成长的机会，如果没有大家平时的帮助，就不会有本书的诞生。

由于编写时间仓促，水平和经验有限，书中难免有欠妥和不足之处，恳请读者批评指正。

郑天民

2018年1月于杭州钱江世纪城

读者服务热线：(010) 81055256 印刷质量反馈：(010) 81055315

反盗版热线：(010) 81055315



非卖品！！严禁（售卖和上传互联网平台）！！  
仅供对书籍质量进行鉴定甄别！为是否购买正版实体书提供依据！！

# 目 录

## 第一篇 直面微服务 1

### 第1章 直面微服务架构 2

1.1 分布式系统	3
1.1.1 单块系统的问题	3
1.1.2 分布式系统的基本特征	6
1.2 微服务架构	8
1.2.1 微服务的概念	9
1.2.2 微服务架构基础	10
1.2.3 微服务架构与现有架构体系对比	12
1.3 构建微服务架构的系统方法	14
1.3.1 服务模型	15
1.3.2 实现技术	15
1.3.3 基础设施	16
1.3.4 研发过程	16
1.4 微服务架构的优势	16
1.4.1 技术优势	16
1.4.2 业务与组织优势	18
1.5 微服务架构的挑战	20
1.5.1 技术架构挑战	20
1.5.2 研发过程挑战	21

### 1.6 实施微服务架构 22

#### 1.6.1 微服务架构实施前提 22

#### 1.6.2 微服务架构实施模式 23

#### 1.7 本章小结 23

## 第二篇 服务建模 24

### 第2章 服务建模方法 25

#### 2.1 服务分类 25

##### 2.1.1 服务的基本类别 26

##### 2.1.2 服务与业务 29

#### 2.2 服务模型 30

##### 2.2.1 服务的概念模型 31

##### 2.2.2 服务的统一表现形式 32

#### 2.3 服务边界 33

##### 2.3.1 识别业务领域及边界 33

##### 2.3.2 界限上下文 36

##### 2.3.3 服务边界划分的原则 41

#### 2.4 服务数据 41

##### 2.4.1 规范化数据模型的问题 41

##### 2.4.2 数据去中心化 42

#### 2.5 本章小结 47



<b>第3章 服务拆分与集成</b>	<b>48</b>	4.3.1 服务器端负载均衡	92
3.1 服务拆分	49	4.3.2 客户端负载均衡	93
3.1.1 服务拆分的维度	49	4.3.3 负载均衡算法	94
3.1.2 服务拆分的策略	50	4.4 服务路由	95
3.1.3 管理服务的依赖关系	53	4.4.1 直接路由	95
3.1.4 管理服务的数据	56	4.4.2 间接路由	96
3.1.5 管理事务的边界	59	4.4.3 路由规则	96
3.2 服务集成	61	4.5 API 网关	97
3.2.1 系统集成基础	61	4.5.1 网关的作用	98
3.2.2 RPC	62	4.5.2 网关的功能	99
3.2.3 REST	64	4.6 配置管理	100
3.2.4 消息传递	70	4.6.1 配置中心模型	101
3.2.5 服务总线	72	4.6.2 分布式协调机制	102
3.2.6 数据复制	74	4.7 本章小结	104
3.2.7 客户端集成	76	<b>第5章 微服务架构关键要素</b>	<b>105</b>
3.2.8 外部集成	78	5.1 服务治理	106
3.3 本章小结	80	5.1.1 服务注册中心	106
<b>第三篇 服务实现</b>	<b>81</b>	5.1.2 服务发布与注册	109
<b>第4章 微服务架构基础组件</b>	<b>82</b>	5.1.3 服务发现与调用	110
4.1 服务通信	82	5.1.4 服务监控	111
4.1.1 网络连接	82	5.2 数据一致性	113
4.1.2 IO 模型	83	5.2.1 分布式事务	113
4.1.3 可靠性	85	5.2.2 CAP 理论与 BASE 思想	116
4.1.4 同步与异步	85	5.2.3 可靠事件模式	118
4.2 事件驱动	88	5.2.4 补偿模式	124
4.2.1 基本事件驱动架构	88	5.2.5 Sagas 长事务模式	126
4.2.2 事件驱动架构与领域模型	89	5.2.6 TCC 模式	127
4.3 负载均衡	92	5.2.7 最大努力通知模式	133
		5.2.8 人工干预模式	135
		5.2.9 数据一致性模式总结	135



5.3 服务可靠性	136	6.4.3 服务实现	188
5.3.1 服务访问失败的原因	136	6.5 本章小结	193
5.3.2 服务失败的应对策略	138	<b>第 7 章 微服务架构管理体系</b>	<b>194</b>
5.3.3 服务容错	139	7.1 服务测试	194
5.3.4 服务隔离	140	7.1.1 微服务测试的维度	195
5.3.5 服务限流	143	7.1.2 微服务测试实现方法	198
5.3.6 服务降级	145	7.1.3 消费者驱动的契约测试	200
5.4 本章小结	148	7.2 服务交付与部署	205
<b>第 6 章 微服务架构实现技术</b>	<b>149</b>	7.2.1 微服务交付管理	205
6.1 微服务架构实现技术选型	149	7.2.2 基于 Docker 部署微服务	209
6.1.1 技术选型的参考标准	150	7.3 服务监控	219
6.1.2 微服务实现框架对比	152	7.3.1 日志聚合	220
6.2 Spring Boot	153	7.3.2 服务跟踪	224
6.2.1 Spring Boot 概览	154	7.4 服务安全	227
6.2.2 Spring Boot 核心原理	155	7.4.1 通用安全性技术	228
6.3 Spring Cloud	157	7.4.2 安全性协议	230
6.3.1 Spring Cloud 概览	157	7.4.3 微服务中的安全性设计	235
6.3.2 Spring Cloud Netflix Eureka 与服务治理	159	7.5 本章小结	237
6.3.3 Spring Cloud Netflix Ribbon 与负载均衡	165	<b>第四篇 服务转型</b>	<b>239</b>
6.3.4 Spring Cloud Netflix Hystrix 与服务容错	168	<b>第 8 章 向微服务架构转型</b>	<b>240</b>
6.3.5 Spring Cloud Netflix Zuul 与 API 网关	177	8.1 微服务架构转型过程与方法	241
6.3.6 Spring Cloud Config 与配置 中心	180	8.1.1 调整架构的技术	242
6.4 案例分析	184	8.1.2 微服务架构与现有系统	245
6.4.1 服务建模	184	8.1.3 微服务实施最佳实践	251
6.4.2 服务架构设计	186	8.2 微服务架构与研发过程转变	256
		8.2.1 产品管理转变	256
		8.2.2 组织架构转变	259
		8.2.3 研发文化转变	262



8.3 微服务架构转型案例分析	264	8.3.5 微服务架构改造第三阶段	280
8.3.1 系统描述	264	8.3.6 微服务架构改造第四阶段	285
8.3.2 微服务架构改造整体方案	268	8.4 本章小结	290
8.3.3 微服务架构改造第一阶段	268	参考文献	291
8.3.4 微服务架构改造第二阶段	273		
服务集成			
3.2.1 系统架构	202		
3.2.2 基于 Docker 部署策略	209		
3.2.3 REST	219		
3.2.4 消息传递	220		
3.2.5 服务总线	224		
3.2.6 数据复制	227		
3.4.1 通用安全框架	228		
3.4.2 安全协议	230		
3.4.3 微服务中的安全性	232		
本章小结	237		
第三篇 服务实现	18		
第四章 微服务架构基础组件	28		
8.1 微服务架构转型案例	281		
8.1.1 案例背景	281		
8.1.2 案例目标	281		
8.1.3 案例实施	281		
8.1.4 案例总结	281		
8.2 微服务架构基础组件	281		
8.2.1 产品管理	281		
8.2.2 事件驱动	281		
8.2.3 负载均衡	281		



# 第一篇 直面微服务

## ◆ 本篇内容

本篇从微服务的基本概念出发，阐述微服务架构的方方面面。我们经历了从单块系统到分布式系统的演变，而微服务架构基于分布式系统，也借助了面向服务架构和企业服务总线的设计理念，并做了改进和优化，从而形成一种全新的架构体系。

微服务架构一方面具备技术、业务和组织上的优势，另一方面也在技术架构和研发过程上存在较大挑战。微服务架构的实施需要具备一定前提，构建微服务架构是一项系统工程，涉及服务建模、实现技术、基础设施和研发过程等各个维度；也需要根据现有系统的具体情况采用合适的实施模式。

本篇共有一章，作为开篇总领全书后续章节。



## 第1章

# 直面微服务架构

随着近年来软件行业（尤其是互联网行业）飞速发展，一方面以电子商务、O2O、移动医疗、在线教育等为代表的互联网和互联网+应用对软件行业的业务结构和体系造成巨大冲击，另一方面，快速的业务更新和产品迭代也给系统开发过程和模式带来新的挑战。业务需求层出不穷且不断变化、技术发展和创新日新月异、团队规模从无到有快速扩张、系统的复杂性以及对行业变化的快速应变能力等成为软件开发的核心问题。围绕这些来自于外部与内部的压力和动力，如何更为合理地划分系统和团队边界、如何更加有效地组织系统开发过程、如何通过技术手段识别和消除开发过程中的浪费成为广大软件开发和技术管理人员需要思考的命题。

针对这些命题，我们的思路首先是根据业务划分系统和领域的边界，对一个大而全的系统进行分而治之，通过一些功能边界明确、业务高度抽象的模块或组件之间的组装去形成更大的业务体系。其次，在系统和领域内部，同样需要对业务体系进行合理建模。通过建立服务体系对服务进行一定拆分和集成，从而降低业务实现的复杂性，并提高服务交互的灵活性。再次，通过服务拆分和集成的手段也可以推动研发团队组织架构的优化，并促进系统持续交付工作的有效开展。围绕这些解决问题的思路，微服务架构（Microservices Architecture）为我们提供了一种具体的解决方案。

所谓微服务（Microservices），就是一些具有足够小的粒度、能够相互协作且自治的服务体系。每个微服务都比较简单，仅关注于完成一个具体业务功能并能很好地完成该功能，而这里的功能代表都是一种业务能力。构建微服务体系需要一套完整的方法论和工程实践，而微服务架构代表的是实现微服务体系的架构模式，即为我们提供了这些方法论和工程实践。通过微服务架构，软件开发过程能够得到改善，开发效率能够得到提高，从而创造更为优秀的产品和用户满意度。微服务架构的提出代表着一种新的架构设计风格和模式，从这个角度讲，微服务架构需要我们理解、学习并应用到日常开发过程中去。但是微服务架构又不是一种完全打破现有技术体系、从无到有所诞生的替代性架构，而是在现有面向服务架构、企业服务总线等思想和技术体系的基础上，伴随着持续交付、虚拟化和容器技术的发展自然而然产生的一种软件设计和架构模式。

微服务架构的基础是分布式系统。本章从分布式系统出发，围绕微服务架构的基本概念展



开, 讨论微服务架构与现有架构体系之间的对比, 并给出构建微服务架构的系统方法。同时, 微服务架构有其优势, 但在实施过程中也会面临各种各样的挑战, 本章对微服务架构的这些特点也做了分析。

## 1.1 分布式系统

微服务架构首先表现为一种分布式系统 (Distributed System), 而分布式系统是对传统单块系统 (Monolith System) 的一种演进。为了更好地理解和掌握微服务架构的特点, 本节对单块系统和分布式系统做简要介绍。

### 1.1.1 单块系统的问题

在软件技术发展过程的很长一段时间内, 软件系统都表现为一种单块系统。时至今日, 很多单块系统仍然在一些行业和组织中得到开发和维护。所谓单块系统, 简单讲就是把一个系统所涉及的各个组件都打包成一个一体化结构并进行部署和运行。在 Java EE 领域, 这种一体化结构很多时候就体现为一个 WAR 包, 而部署和运行的环境就是以 Tomcat 为代表的各种应用服务器。图 1-1 所示的就是一个典型的单块系统。我们可以看到在应用服务器上同时运行着面向用户的 Web 组件、封装业务逻辑的 Service 组件和完成数据访问的 DAO (Data Access Object, 数据访问对象) 组件。这些组件都作为一个整体进行统一的开发、部署和维护。

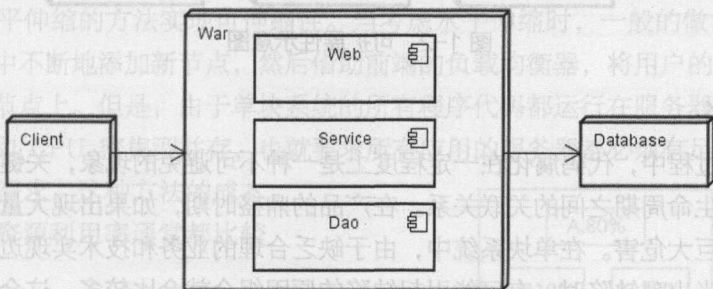


图 1-1 单块系统示意图

显然, 图 1-1 所示的单块系统有其存在和发展的固有优势。当团队规模并不是太大的时候, 一个单块应用可以由一个开发者团队进行独立维护。该团队的成员能够对单块应用快速学习、理解和修改, 因为其结构非常简单。同时, 因为单块系统的表现形式就是一个独立的 WAR 包, 想要对它进行集成、部署以及实现无状态集群相对也比较简单, 通常只要采用负载均衡机制并运行该单块系统的多个实例就能达到系统伸缩性要求。



但在另一方面，随着公司或者组织业务的不断扩张、业务结构的不断变化以及用户量的不断增加，单块架构的优势已逐渐无法适应互联网时代的快速发展，面临着越来越多的挑战。在使用单块系统时，我们不得不面对以下问题。

## 1. 业务复杂度

对于大多数系统而言，架构设计是为了满足业务需求的。衡量架构好坏与否的一个重要方面是看其面对复杂业务变更时所应该具有的灵活性，也就是我们通常所说的可扩展性（Extensibility）。可扩展性是指系统在经历不可避免的变更时所具有的灵活性，与针对提供这样的灵活性所要付出的成本进行平衡的能力。所谓可扩展，可扩展的是业务。即当往 SystemA 中添加新业务 NewSubSystem 时，如果不需要改变原有的各个子系统而只需把新业务封闭在一个新的子系统中就能完成整体业务的升级，我们就可以认为系统具有较好的可扩展性，可扩展性示意图如图 1-2 所示。显然，单块系统不具备良好的可扩展性，因为对系统业务的任何一处进行修改，都需要重新构建整个系统并进行发布。单块系统内部没有根据业务结构进行合理的业务拆分是导致其可扩展性低下的主要原因。

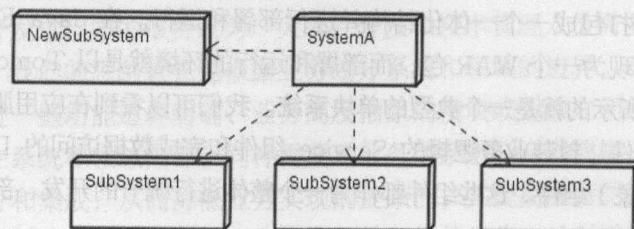


图 1-2 可扩展性示意图

## 2. 代码腐化

在软件开发过程中，代码腐化在一定程度上是一种不可避免的现象，关键是腐化的时间和程度与整个产品生命周期之间的关联关系。在产品的鼎盛时期，如果出现大量的代码腐化会对产品的发展带来巨大危害。在单块系统中，由于缺乏合理的业务和技术实现边界，随着产品业务功能的增多，当出现缺陷时，有可能引起缺陷的原因组合就会比较多，这会导致分析、定位和修复缺陷的成本相应增高，也就意味着缺陷的平均修复周期可能会花费更长时间，从而影响到产品的正常迭代和演进。同时，随着功能不断叠加，单块系统的代码结构也日益复杂，在开发人员对全局功能缺乏深度理解的情况下，修复一个缺陷的同时还有可能引入其他的缺陷，在很多技术团队并不具备完善的自动化测试机制的客观条件下，很可能导致问题越修越多的不良循环。



### 3. 团队问题

互联网行业由于产品价值与市场时机密切相关，普遍崇尚快速试错和迭代发布，很多公司或组织会在比较短的时间内扩充产品功能和开发团队，也就意味着对于过程资产建设和人才培养等方面并不会投入太多的成本，这就要求新加入团队的成员能够快速融入团队并进行代码开发和维护。然而，由于在单块系统中所有的业务和代码在很大程度上无序地混合在一起，存在大量错综复杂的业务和代码结构、由于历史原因所造成的迥然不同的开发风格以及看似复杂但已经不被使用的遗留代码，使得新员工了解行业背景、熟悉应用程序业务、配置本地开发环境等看似简单的任务都变得并不简单。

另外，单块系统的集中式管理方式，使得系统内部的技术体系和开发方式很难得到扩展。随着应用程序的复杂性逐渐增加以及功能越来越多，如果团队希望尝试引入新的框架和技术，或者对现有技术栈升级，通常都会面临不小的风险。也即意味着初始的技术选型严重限制了单块系统将来采用不同开发语言或框架的能力。但互联网行业中技术体系变化和业务体系变化一样快速，技术体系的无法扩展和升级也会导致现有团队中不同出身和开发背景的成员对系统代码产生一种开发惰性，也无法吸引到优秀的开发人员。

### 4. 伸缩性问题

前面讲到单块系统的可扩展性很差，实际上它的可伸缩性同样很有问题。所谓可伸缩（Scalability），伸缩的是性能，即当系统性能出现问题时，如果我们只需要简单添加应用服务器等硬件设备就能避免系统出现性能瓶颈，那么该系统无疑具备较高的可伸缩性。通常，我们会考虑采用水平伸缩的方法实现可伸缩性。当考虑水平伸缩时，一般的做法是建立一个集群，通过在集群中不断地添加新节点，然后借助前端的负载均衡器，将用户的请求按照某种算法分配到不同的节点上。但是，由于单块系统的所有程序代码都运行在服务器上的同一个进程中，内存密集型和 CPU 密集型并存，也就要求所有应用的服务器都必须有足够的内存和强劲的 CPU 来满足需求。这种方法的成本会比较高，而且资源利用率通常都比较低下。

以图 1-3 为例，单块系统中的组件 A 的负载已经达到了 80%，也就是到了不得不对系统运行能力进行扩容的时候。但同一系统的其他两个组件 B 和 C 的负载还没有达到其处理能力的 20%。由于单块系统中的各个组件是打包在同一个 WAR 包中的，因此通过添加一个

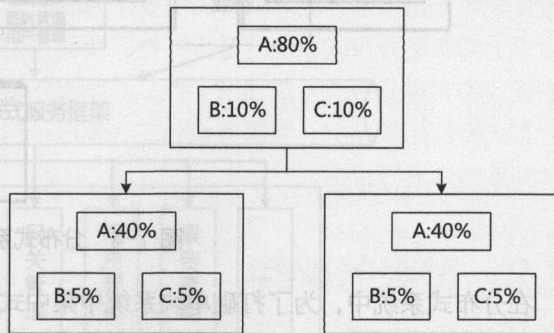


图 1-3 单块系统的伸缩性问题



额外的系统实例虽然可以将需要扩容组件的负载降低一半，但是显然其他组件的利用率变得更低，造成资源浪费。另外，对于那些需要保持类似会话（Session）数据的需求而言，扩容之后的运行机制在如何保持各个服务器之间数据的一致性上，也存在较大的实现难度。

针对以上集中式单块系统所普遍存在的问题，基本的解决方案就要依赖于分布式系统的合理构建。

### 1.1.2 分布式系统的基本特征

分布式系统，是指硬件或软件组件分布在不同的网络计算机上，彼此之间仅仅通过消息传递进行通信和协调的系统。我们从这个定义中可以看出分布式系统包含两个区别于单块系统的本质性特征：一个是网络，分布式系统的所有组件都位于网络之中，对于互联网应用而言，则位于更为复杂的互联网环境中；另一个是通信和协调，与单块系统不同，位于分布式系统中的各个组件只有通过约定、高效且可靠的通信机制进行相关协作才能完成某一项业务功能。这是我们在设计和实现分布式系统时首先需要考虑的两个方面。图 1-4 所示的就是从软件开发视图出发得到的一个典型的分布式系统，包含了分布式服务、消息中间件和分布式缓存等常见的用于构建分布式系统的技术实现方式。显然，这些工具位于一个封闭或开放的网络环境中，相互之间通过服务的注册和发现、消息传递、数据的缓存共享等机制完成协作。

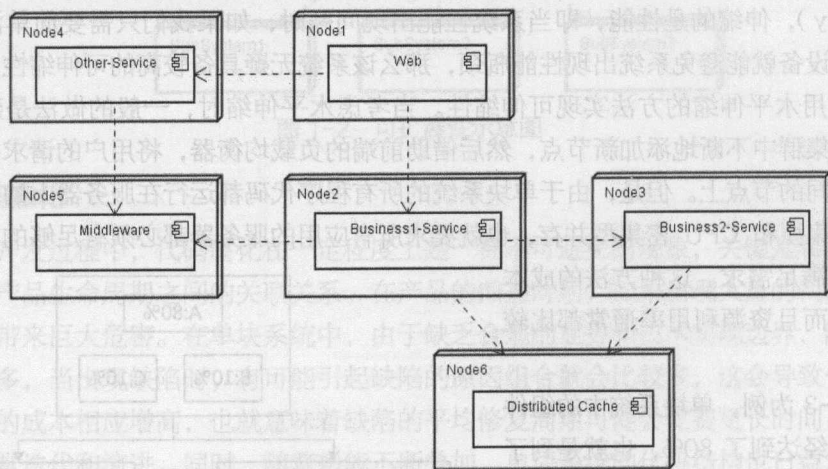


图 1-4 分布式系统示意图

在分布式系统中，为了打破单块系统中集中式的系统架构，我们引入系统拆分思想和实践。拆分的需求来自组织结构变化、交付速度、业务需求以及技术需求所引起的变化。一般认为系统拆分的基本思路有两种，即纵向（Vertical）拆分和横向（Horizontal）拆分。



所谓纵向拆分，就是将一个大应用拆分为多个小应用，如果新业务较为独立，那么就将其设计部署为一个独立的应用系统即可。如图 1-5 所示，我们可以将移动医疗系统中的预约挂号业务拆分成订单、医院和用户等独立业务子系统。纵向拆分关注于业务，通过梳理产品线，将内聚度较高的相关业务进行剥离从而形成不同的子系统。

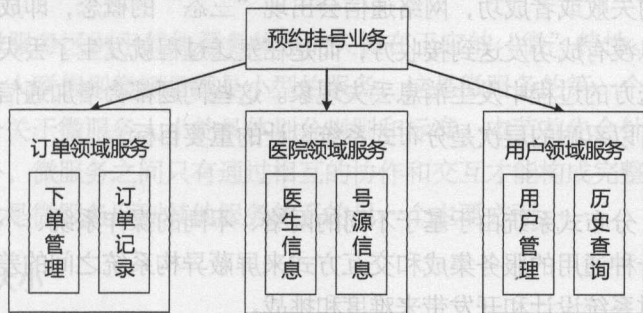


图 1-5 分布式纵向拆分示意图

相对于纵向拆分的面向业务特性，横向拆分更多地关注于技术。所谓横向拆分，就是通过将可以复用的业务拆分出来，独立部署为分布式服务，调用这些分布式服务，构建复杂的新业务。所以，横向拆分的关键在于识别可复用的业务，设计服务接口并规范服务依赖关系。横向拆分的基本实现方式是构建分布式服务体系，图 1-6 是对图 1-5 所示的预约挂号业务进行横向拆分的结果。可以看到，当我们把订单、医生、号源和用户等业务抽象成独立的垂直化服务，并在各个服务上层实现分布式环境下的调用和管理框架，系统的业务就可以转变为一种排列组合的构建方式。如基于订单和支付服务，我们可以构建出业务 1，而业务 2 可能只依赖于医院和用户管理服务。分布式服务框架提供了一种按需构建的机制，在保证各个分布式服务的技术、团队、交付独立发展的前提下，确保业务整合的灵活性和高效性。

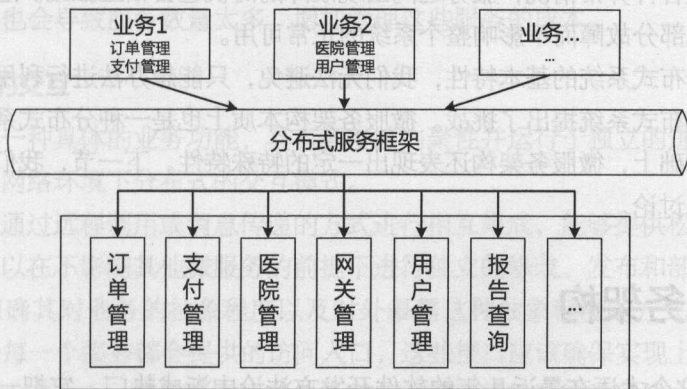


图 1-6 分布式横向拆分示意图



然而，分布式系统相较于单块系统而言具备优势的同时，也存在一些不得不考虑的特性，包括但不限于以下几点。

- 网络传输的三态性

构建分布式系统依赖网络通信，而网络通信表现为一个复杂且不可控的过程。相比于单机系统中函数式调用的失败或者成功，网络通信会出现“三态”的概念，即成功、失败与超时。由于网络原因，消息没有成功发送到接收方，而是在发送过程就发生了丢失现象；或者接收方处理后，响应给发送方的过程中发生消息丢失现象。这些问题都会增加通信的代价。如何使通信的代价降到用户可以忍耐的层次是分布式系统设计的重要目标。

- 异构性

相较单块系统，分布式系统由于基于不同的网络、不同的操作系统、不同的软件实现技术体系，必须要考虑一种通用的服务集成和交互方式来屏蔽异构系统之间的差异。异构系统之间的不同处理方式会对系统设计和开发带来难度和挑战。

- 负载均衡

在集中式系统中，各部件的任务明确。由于分布式系统是多机协同工作的系统，为了提高系统的整体效率和吞吐量，必须考虑最大程度发挥每个节点的作用。负载均衡是保证系统运行效率的关键技术。

- 数据一致性

在分布式系统中，数据被分散或者复制到不同的机器上，如何保证各台主机之间的数据一致性将成为一个难点。因为网络的异常会导致分布式系统中只有部分节点能够正常通信，从而形成了网络分区（Network Partition）。

- 服务的可用性

分布式系统中的任何服务器都有可能出现故障，且各种故障不尽相同。而运行在服务器上的服务也可能出现各种异常情况，服务之间出现故障的时机也会相互独立。通常，分布式系统要设计成允许出现部分故障而不影响整个系统的正常可用。

以上问题是分布式系统的基本特性，我们无法避免，只能想办法进行利用和管理，这就给我们设计和实现分布式系统提出了挑战。微服务架构本质上也是一种分布式系统，但在遵循通用分布式特性的基础上，微服务架构还表现出一定的特殊特性。下一节，我们将围绕微服务架构的特殊特性展开讨论。

## 1.2 微服务架构

微服务架构这个术语在最近几年的软件开发方法论中渐成热门，它把一种特定的软件应用设计方法描述为能够独立部署的服务套件。目前对于微服务架构还缺乏统一的标准化定义，



但其在业务建模、服务集成、数据去中心化等方面上的抽象和提炼已得到普遍认可。在讨论微服务架构之前，我们先介绍一下微服务的概念。

## 1.2.1 微服务的概念

顾名思义，微服务区别于其他服务体系的关键在于它的“微”特性。“微”是“小”的同义词，所以容易让人联想到微服务都是小型的服务，这是微服务的第一个特性。然而，目前业界并没有给出一个关于微服务大小的具体划分规则和标准，本节首先会针对微服务的“微”特性展开讨论。另外，微服务之间只有通过相互的协作和交互才能构成完整的服务体系，而这种协作和交互机制也是微服务区别其他服务体系的另一主要方面。

### 1. 微服务的大小

通常我们可以使用代码行数、开发时间来给微服务的大小添加一些约束条件，即满足一定代码行数或开发时间之内的服务才能称为微服务。这些当然都是可衡量的量化标准，但针对具体业务场景往往不能简单通过这些指标就能得到合理的微服务划分结果。

我们认为微服务应该足够小，小到专注于某一个具体业务功能，也就是说首先应该保证每个微服务是具有业务独立性的工作单元。在考虑微服务时，我们需要根据业务边界来确定服务的边界，这样就可以把该业务相关的内容都集中在微服务内部，从而体现了服务的高内聚性。

同时，服务的大小还跟团队的规模和工作方式有较大关系。如果一个系统大到不能够由一个独立团队进行开发和维护，那么将其拆分成适合这个团队开发能力的服务无疑是合理的做法。而如果这个团队本身的规模已经大到需要进行进一步分工和协作，那么先把团队进行拆分，再根据团队拆分的结果进行服务拆分也是一项最佳实践。

然而，我们还需要认识到服务过小可能带来的问题。服务越小，其独立性和高内聚性能够带来优势，但是也会导致服务数量太多，增加管理这些服务的成本。

### 2. 微服务的交互

微服务代表一种具体的业务功能，具有高度的内聚性并运行于独立的进程中。因此，微服务之间的交互是网络环境下分布式的交互模式。

微服务之间通过远程调用或消息传递的方式进行相互集成，能够提供松耦合的架构体系，每个微服务都可以在不影响其他微服务的前提下进行独立的修改、发布和部署。对于微服务而言，我们应该明确其对业务的抽象程度以及对外暴露这种抽象程度的方式。通常，服务接口（Interface）是每一个服务都会提供的访问入口，这些接口应该确保实现上的技术无关性。

同时，在技术实现上，我们也需要确保微服务之间采用轻量级的通信方式进行交互，因为类似采用长连接方式进行通信的过程会导致服务之间不能很好地解耦，一旦出现问题，耦合性



会触发异常,在各个微服务之间进行扩散,从而导致整个服务体系不可用。这是我们在采用微服务时所需要极力避免的场景。对于轻量级通信机制而言,通常建议采用 HTTP 协议让服务之间的通信变得标准化和无状态化。

微服务的大小和服务之间的交互方式构成了基本的微服务体系结构,分别代表了微服务高内聚、低耦合的基本特性。在本书后续章节中,我们会集中讨论服务的边界划分原则和方法、服务管理的要求和实践以及微服务之间集成的各种技术体系。

微服务是一种服务体系,关于如何构建这种服务体系,在设计和实现上需要特定的方法论,而微服务架构就是用来构建微服务的架构模式,为我们提供了具体的方法论以及相关的工程实践。

## 1.2.2 微服务架构基础

微服务架构是一种架构模式,区别于其他系统架构的构建方式和技术方案,微服务架构具有其固有点,这些特点也为我们在使用微服务架构进行系统设计的过程提供了主要的切入点。

### 1. 微服务架构特点

马丁·福勒(Martin Fowler)指出<sup>[1]</sup>,微服务架构具有以下特点。

#### (1) 服务组件化

组件(Component)是一种可独立替换和升级的软件单元。在我们日常开发过程中可能会设计和使用的很多组件。这些组件可能服务于系统内部,也可能存在于系统所运行的进程之外。而服务就是一种进程外组件,服务之间利用诸如远程过程调用(Remote Procedure Call, RPC)的通信机制完成交互。服务组件化的主要目的是服务可以独立部署。如果你的应用程序是由很多组件组成,并且这些组件运行在同一个进程中,那么对任何一个组件的改变都将导致必须重新部署整个应用程序。但是,如果你把应用程序拆分成很多服务,显然通常情况下,你只需要重新部署那个改变的服务。在微服务架构中,每个服务运行在其独立的进程中,服务与服务之间采用轻量级通信机制互相沟通。

#### (2) 按业务能力组织服务

当寻找把一个大的应用程序进行拆分的方法时,研发过程通常都会围绕产品团队、UED 团队、APP 前端团队和服务器端团队而展开。这些团队也就是通常所说的职能团队(Function Team)。当使用这种标准对团队进行划分时,任何一个需求变更,无论大小,都将导致跨团队协作,从而增加沟通和协作成本。而微服务架构下的划分方法有所不同,它倾向于围绕业务功能的组织来分割服务。这些服务面向具体业务结构,而不是面向某项技术能力。因



此，团队是跨职能的（Cross-Functional）的特征团队（Feature Team），包含用户体验、项目管理和技术研发等开发过程所要求的所有技能。每个服务都围绕着业务进行构建，并且能够被独立部署到生产/类生产环境。

### （3）去中心化

服务集中治理的一种好处是在单一平台上进行标准化，但采用微服务的团队更喜欢不同的标准。把整体式框架中的组件拆分成不同的服务，我们在构建这些服务时就会有更多的选择。对具体的一个服务而言，应该根据业务上下文，选择合适的语言、工具进行构建。

另一方面，微服务架构也崇尚对数据进行分散管理。当整体式的应用使用单一逻辑数据库进行数据持久化时，通常选择在应用的范围内使用一个数据库。然后，微服务让每个服务管理自己的数据库，无论是相同数据库的不同实例，或者是不同的数据库系统。

### （4）基础设施自动化

许多使用微服务架构的产品或者系统，它们的团队拥有丰富的持续集成（Continue Integration）和持续交付（Continuous Delivery）的经验。团队使用微服务架构构建软件需要更广泛的依赖基础设施自动化技术。

微服务同样需要考虑服务容错性设计等分布式系统所需要考虑的问题，我们对以上特点进行总结和提炼，认为微服务具备业务独立、进程隔离、团队自主、技术无关轻量级通信和交付独立性等“微”特性。

## 2. 微服务架构设计的切入点

微服务架构设计的首要的切入点在于服务建模，尽可能明确领域的边界。我们可以充分利用领域驱动设计（Domain Driven Design, DDD）方法，通过识别领域/子域中的模块和服务、判断这些模块和服务是否独立、考虑提升某些模块和服务的层次并建立充血领域模型，从而明确各个界限上下文（Boundary Context）之间的边界。

微服务架构设计第二个切入点就是服务之间的集成方式，即需要保证集成方式的技术无关性，不要选择对服务具体实现有技术性限制的集成技术。采用技术无关的集成接口，充分融合技术多样性，意味着我们在特定场景下不应该使用类似 Alibaba Dubbo (<http://dubbo.io/>) 这种采用私有协议、重量级的通信框架，而应该尽可能使用类如 RESTful 的轻量级通信方式进行服务集成，确保服务易于使用，消费方可以使用多种技术实现集成。

微服务架构设计的第三个切入点在于服务的部署，独立部署单个服务而不需要修改其他服务。同时，由于服务数量大，修改和发布的频率也可能很高，接口变化管理上通常采用逐步迁移的方案。而在接口的发布上，常见的 API 网关（Gateway）等模式也会得到广泛应用。



### 1.2.3 微服务架构与现有架构体系对比

微服务其实并不是凭空产生，而是有其历史的渊源。在微服务概念被正式提出之前，我们会经常使用面向服务架构（Service Oriented Architecture, SOA）来构建分布式服务系统。SOA 只是提出了一种架构设计思想，但没有给出标准的参考实现。而早期企业软件开发上也摸索了一套实践方式，即企业服务总线（Enterprise Service Bus, ESB）。本节通过与 SOA 和 ESB 这两种架构模式的对比来更加深入地分析微服务架构。

#### 1. 微服务架构与 SOA

在山姆·纽曼（Sam Newman）的《Building Microservices》<sup>[3]</sup>一书中，作者认为微服务架构是 SOA 的一种实践方式，正如 XP（Extreme Programming, 极限编程）或 Scrum 是敏捷软件开发的实践方式。在介绍两者之间的区别和联系之前，我们先来看一下 SOA 的特点。

SOA 中，粗粒度的服务接口分级、松耦合的服务调用关系、标准化的服务接口、精确定义的服务契约、服务接口设计管理、支持各种消息模式等是服务的主要特征。SOA 从分层上看，存在三个主要的边界，在面向用户的门户（Portal）层与提供业务功能的服务层之间，还存在一个层次用于服务之间的集成（Integration）和编排（Orchestration）。显然，各个服务通过集成和编排能够组合成更为丰富的业务功能和体系，而集成和编排能够实现的前提正是依赖于服务自身所具备的这些特征。图 1-7 所示的是一种典型的基于 SOA 的服务交互方式。

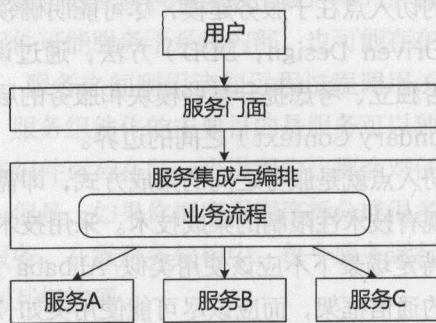


图 1-7 SOA 的服务交互方式

在服务实现上，因为需要考虑到服务之间的集成和编排，采用 SOA 时需要引入一些复杂的技术体系。以在 SOA 中被广泛使用的 Web Service 为例，复杂的交互协议使得采用 SOA 时需要付出巨大的代价。SOA 本身崇尚简单和互操作性，但随着事务、安全性等一系列扩展功能的出现，Web Service 变成了一个庞大的 WS-\*协议栈，反而阻碍了服务之间的交互。

微服务架构与 SOA 无疑是有关联的，两者都是将业务系统拆分成各个服务并通过合适的



技术完成服务的集成。微服务架构与 SOA 的最主要区别在于架构面向的目标。SOA 通常面向的目标是企业级应用，通过 SOA 我们寄希望于将多个系统整合到一起，从而消除信息孤岛。而微服务架构则更多地关注于一个独立系统的架构，可以认为是传统模块化技术的一种替代方案。

微服务架构在服务之间同样需要实现集成，但在集成方式上强调使用轻量级的通信技术，同时去除服务编排功能。一个微服务可以调用其他微服务并在服务内部实现编排操作。这样编排操作作为微服务的内嵌功能不会暴露在集成层，从而降低服务之间的通信成本。另一方面，微服务之间的集成模式也与 SOA 中完全不同。由于微服务是面向业务、具备领域特性的独立服务，我们可以在用户操作层面直接进行服务集成。微服务之间的交互关系如图 1-8 所示，我们可以看到微服务可以直接面向用户，且微服务之间也可以直接进行交互，服务与服务之间没有 SOA 中的集中式编排机制。

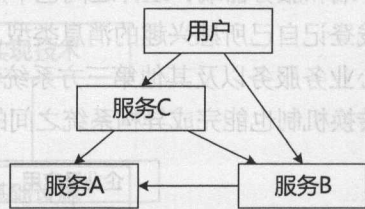


图 1-8 微服务架构的服务交互方式

当然，并不是说，微服务架构不能使用统一的集成和编排机制。在需要对很多服务进行统一的权限管理、日志记录、请求管理等场景时，我们也可以在微服务的访问入口上添加统一的拦截。在微服务架构中，我们通常使用 API 网关（Gateway）实现拦截处理。关于 API 网关的讨论我们将放在本书第 4 章。

由于 SOA 和微服务架构比较容易混淆，我们再从应用范围、灵活性、组织性和部署等方面对两者做一个对比总结。

#### （1）应用范围

在应用范围上，SOA 是一种企业级的、面向大范围 and 统一化的服务化架构，而微服务架构通常用于某一个项目或产品，并不强调大而全的服务集成需求。

#### （2）灵活性

SOA 通过服务编排机制实现灵活性，而微服务架构的灵活性则来自于快速的开发和部署以及服务之间的独立性。

#### （3）组织性

通常，SOA 中的服务由不同组织中的职能团队实现，而微服务则更强调跨职能团队机制，同一项目或产品中具备各个职能的人员或团队，共同实现微服务。

#### （4）部署

在 SOA 中，部分服务仍然以单块系统的方式进行部署，而独立进程部署是微服务架构的基本特征，所有的服务都能独立部署。



## 2. 微服务架构与 ESB

消息总线（Message Bus）思想的理论基础是消息驱动的编程方法和计算机硬件总线概念。系统组件之间通过总线来进行通信，可以支持组件的分布式存储和并发运行。总线是系统的连接件，负责消息的分派、传递和过滤，并返回处理结果。总线思想在软件设计过程中应用广泛，并形成了一整套完整的企业服务总线（Enterprise Service Bus, ESB）技术体系。图 1-9 展示了一种典型的服务总线应用场景。我们可以看到在 ESB 中，系统组件并不严格区分客户端和服务端，组件之间也不是通过消息通道进行直接交互，而是挂载在消息总线上，向总线登记自己所感兴趣的消息类型。通过这种方式，图 1-9 所示的企业级应用、遗留系统、核心业务服务以及其他第三方系统都可以通过总线串联起来。同时，ESB 所提供的资源适配和转换机制也能完成异构系统之间的无缝集成。



图 1-9 ESB 架构

在消息总线中，消息是组件之间唯一的通信方式。根据需要，消息总线对消息具备丰富的预处理功能，包括消息路由（Routing）、消息转换（Transformation）和消息过滤（Filtering）。这些预处理功能消除了数据传递在时间、空间和技术上的耦合，并提供了高度扩展性，但同时也为系统的设计和实现带来了不可避免的复杂度。

微服务架构对于 ESB 的改变在于强化端点（Endpoint）及弱化通道，抛弃了 ESB 过度复杂的业务规则编排、消息路由等功能，并引入了哑管道（Dumb Pipe）的设计理念。这里的终端是指服务本身，而哑管道是通信机制。在微服务架构中，服务作为智能端点（Smart Endpoint），所有的业务逻辑在服务内部处理，而服务间的通信尽可能的轻量化，可以是同步的远程过程调用（RPC），也可以是异步的消息传递系统，它们只作为消息通道，在传输过程中不会附加额外的业务规则。从这点上讲，微服务架构表现得更像管道-过滤器（Pipe-Filter）模式中的过滤器一样，接受请求、处理业务逻辑并返回响应。

## 1.3 构建微服务架构的系统方法

构建微服务架构所需要做的不仅仅是构建服务本身。一个微服务系统的构建过程代表的是



一种组织级别的活动，包括组织的人员架构、研发过程、技术体系和协作文化等多个因素。同样，微服务的运行时环境、错误处理机制和运维实践也是我们需要考虑的内容。本节中我们将针对如何构建微服务架构给出一套完整的系统方法，并围绕这个系统方法引出本书后续章节的内容大纲。

图 1-10 给出了构建微服务架构的系统方法。这套系统方法有助于把所需要做的工作进行分解并形成切入点。我们可以看到整个系统方法包括服务模型、实现技术、基础设施和研发过程等四个方面内容。



图 1-10 微服务架构构建模型

### 1.3.1 服务模型

服务建模是实现微服务架构的第一步，因为微服务架构与 SOA、ESB 等现有技术体系的本质区别就是其服务的粒度和服务本身的面向业务和组件化特性。针对服务建模，我们首先需要明确服务的类别以及服务与业务之间的关系，从而明确服务的概念模型并给出服务的统一表现形式。同时，我们也需要借助于诸如领域驱动设计中的界限上下文和领域事件等技术合理规划微服务的边界，并剥离微服务与数据之间的耦合。服务模型建立最主要的工作是服务的拆分和集成。服务拆分需要考虑拆分的维度、策略并管理服务之间的依赖关系、数据以及边界。而服务的集成则需要考虑在轻量级服务通信的要求下所应当采用的技术实现方式。本书第 2 章和第 3 章将对服务建模做全面介绍。

### 1.3.2 实现技术

微服务的实现技术是构建微服务架构的重点。微服务架构具有分布式架构的基本特征，所以网络通信、事件驱动、服务路由、负载均衡、配置管理等因素同样是实现微服务架构的基础。另一方面，我们也需要考虑微服务架构实现上的一些关键要素，包括服务治理、数据一致性和服务可靠性等内容。最后，通过技术选型，我们将明确构建微服务的具体实现工具和框架。本书第 4、5、6 章将对如何实现微服务架构给出了系统的分析以及相应的技术实现方案。



### 1.3.3 基础设施

本书第 7 章将介绍微服务架构的管理体系，包括服务的测试、服务的部署、服务的监控和服务安全性等内容。本书的主要目标在于阐述微服务的设计和架构，但这些基础设施仍然是微服务架构整张蓝图的重要组成部分。

### 1.3.4 研发过程

微服务架构构建过程中所涉及的关于业务结构、组织架构和研发文化等方面的内容，我们放在本书第 8 章中介绍。这些内容构成了开发团队的整体研发过程，讨论组织架构和软件开发的关系、构建跨职能团队、强调引入变化和敏捷思想有助于更好的落实微服务架构。

基于以上所阐述的关于微服务架构构建的系统方法，我们还将讨论如何使用微服务架构进行遗留系统改造的方法和实践。通过梳理现有架构的改造技术，明确向微服务架构的转型方法，我们尝试并探寻微服务实施的最佳实践。向微服务架构转型是目前很多团队所面临的一个选择，我们也将本书第 8 章中阐述相应的转型方法，希望给这些团队提供有价值的参考。

## 1.4 微服务架构的优势

微服务架构能够为我们带来许多优势，包括技术上的优势、业务结构上的优势以及组织上的优势，本节将对这些优势展开介绍。

### 1.4.1 技术优势

微服务架构的技术优势非常明显，我们通过一个简单示例来说明它对软件开发所带来的变化。试想在一个移动医疗系统中，用户可以通过 APP 进行预约挂号并形成支付订单，另外，我们也希望实现医生和科室的搜索功能从而为用户提供精准查询的用户体验，如图 1-11 所示。预约挂号和搜索对于产品的业务结构而言是非常简单和明确的功能需求，在实现上却可能有很多种不同的方法。

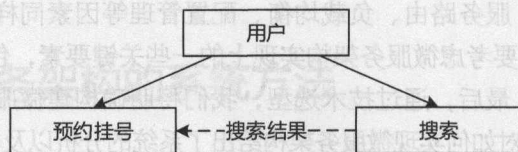


图 1-11 预约挂号和搜索功能



在图 1-11 中，我们发现预约挂号和搜索功能存在一定的依赖关系，因为用户可以先执行搜索操作，然后根据搜索的结果进行预约挂号。这时候，比较容易想到的实现方法是把这两个功能放在同一个模块或子系统中，要修改就一起修改。但是，一旦这种相对较弱的依赖关系在整个系统中开始滋生蔓延时，我们会发现很快就会演变成一种腐化状态。这种腐化状态通常需要使用专门的架构重构才能得到缓解（我们会在第 8 章中讨论这些架构重构方法），而微服务架构的引入能够避免这种现象的发生。在微服务架构中，预约挂号和搜索将成为两个独立的微服务。如果这两种服务之间需要交互，那么我们可以引入明确且轻量级的通信机制。微服务之间的边界起到一种防腐剂的作用，能够降低业务功能之间的依赖关系从而防止架构腐化。

以上的简单案例给出了微服务架构最重要的优势，我们还可以从以下几个方面再进一步展开讨论。

### 1. 组件化方案

微服务架构提供的是一种高内聚、低耦合的组件化方案。组件所能带来的独立性与健壮性微服务都可以具备，但微服务的组件化特征更多表现在对业务的提炼和对边界的思考。使用微服务架构迫使我们使用诸如领域驱动设计的思想去进行策略设计和技术设计，从而为更好地划分业务功能、提取界限上下文和开展系统集成工作提供依据。而这些方法和依据的背后恰恰是我们在架构设计过程中经常会碰到的问题。

### 2. 技术自由度

当系统架构转变成一系列微服务之间的通信和集成时，我们就明白具体实现技术已经不是系统设计和开发的主要约束条件。因为在微服务架构中，各个微服务之间使用的是轻量级的通信机制。所谓轻量级，就是指这些通信机制跟具体实现技术无关，不受限与某一个特定协议或交互媒介。每个微服务高度独立，可以采用适合自身开发团队和技术体系的工具和框架来实现某个微服务，从而为我们提供了宝贵的技术自由度。

### 3. 可扩展性

在对图 1-8 分析时，我们实际上已经讨论了微服务架构与生俱来的可扩展性。通过服务拆分和集成，单个服务在保持通信方式不变的前提下，对其内部功能和技术的改变不会对外部依赖它的服务产生任何影响。结合可扩展性的概念，微服务架构无疑具备这方面的优势。

### 4. 可伸缩性

可扩展性与可伸缩性是两个不同的概念，但高扩展性往往能够带来高可伸缩性，因为可以伸缩的前提是对系统有合理的拆分。当我们明确系统中的运行瓶颈，并把引起这些瓶颈的业务功能构建成独立的微服务，就可以应用服务集群等手段有效加强服务运行时的环境和状态。



## 5. 有效应对遗留系统

微服务是可用于改造遗留系统（Legacy System）的强有力武器。面对遗留系统，一方面该系统的技术体系可能存在设计上的较大缺陷，另一方面则是因为代码量巨大且不容易修改。在代码层次与遗留系统进行直接集成是痛苦且具有挑战性的工作，但是遗留系统以提供接口的方式暴露某些功能入口仍然是一个相对容易实现的过程。一旦获取这些接口，微服务架构就能与之进行通信并完成功能整合。

## 6. 支持持续交付

持续交付通过简单、可重复的流程来确保软件发布过程的可靠性，通常这是通过持续集成和持续交付管道得以实现。微服务作为独立的可部署单元，非常适合使用持续交付，因为每一个服务都可以在不依赖于其他服务的条件下完成发布和部署。基于微服务架构，持续交付管道可以运行得更快，从而加速问题反馈，这是持续交付的主要目标之一。而持续交付的另一个目标是降低系统风险，微服务小而独立，一旦出现问题很容易进行回滚操作。

### 1.4.2 业务与组织优势

康威定律（Conway's Law）<sup>[4]</sup>指出设计系统的组织，其产生的设计和架构等价于组织间的沟通结构。从传统的单块架构到微服务架构实际就是这一定律的一种体现。现在很多开发团队本质上都是分布式的，单块架构的开发、测试、部署协调沟通成本巨大，严重影响效率且容易产生冲突。将单块架构解耦成微服务架构，每个团队开发、测试和发布自己负责的微服务，互不干扰，系统效率得到提升。可见，组织和系统架构之间有一个映射关系：一方面，如果组织结构和文化结构不支持，通常无法成功建立有效的系统架构；另一方面，如果系统设计或者架构不支持，那么你就无法成功建立一个高效的组织。

用更通俗的说法，康威定律就是指组织形式等同系统设计，更直白地说，你想要什么样的系统，就搭建什么样的团队。如果一个组织分成前端团队、Java 后台开发团队、DBA 团队和运维团队，那么就形成了所谓的功能团队，各个小团队各自为政，需要强有力的组织文化和执行力才能确保各个团队之间有效协作；相反，如果你的系统是按照业务边界划分的，大家遵循同一个业务目标把大系统做成小系统、小产品，就容易形成实现微服务架构所需的自治性组织文化。

### 1. 消除过程浪费

软件行业大多数产品开发由时间和成本决定其投入，即一定数量的开发人员通过一定时间的开发工作完成某个具体产品。显然，开发周期的缩短同样意味着开发成本的降低，因此开发



成本与开发周期密切相关。产品开发周期时间与成本之间的关系如图 1-12 所示,可以看到开发时间和开发成本之间并不是一个简单的线性关系,随着开发时间的增长,开发成本增长的趋势越来越明显。出现这样的情况是因为软件开发过程中对范围变更的控制、计划的监控、资源的合理安排都存在风险,且风险随时间演变其发生的概率和造成的影响就越大。图 1-12 所示的就是技术管理面临的一个重要课题,即对于范围已经固定的产品开发任务而言,如何提高开发效率以降低开发时间和成本。

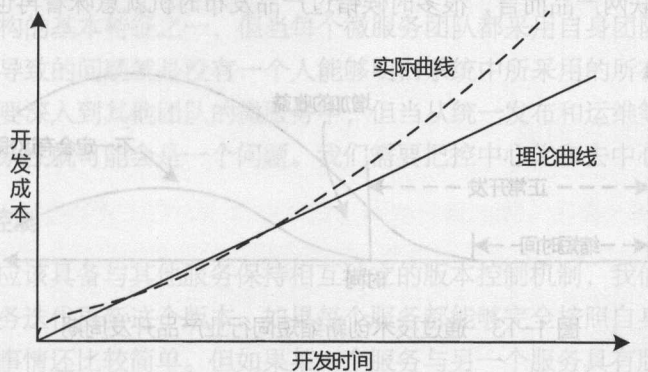


图 1-12 产品开发周期与成本之间的关系

建立高效的研发过程体系可以提高开发效率,通过提高开发效率而节省下来的资源可以再投入到新产品的开发中去,从而使更多的新产品投向市场,或者减少新产品开发的总体成本。对于软件开发这个特定行业而言,减少浪费是提高开发效率的首要切入点。浪费分成纯粹的浪费和必要的浪费,其中纯粹的浪费需要消除,而必要的浪费可以进行压缩<sup>[2]</sup>。

微服务架构的技术特征决定了开发各自微服务的团队之间只需要进行较少的协调工作,这为降低研发过程浪费提供了基础。从组织管理的角度出发,自治性团队较之集中式管理模式下的团队在团队建立和发展上能够得到更好的可扩展性。集中式团队普遍受限于技术上的约束和决策,诸如可用性、性能等非功能性需求都不是由某个业务功能和系统的独立团队所负责,而是需要集中式的、系统级别的实现和管理。微服务架构可以把大型团队打散成小型团队,而小型团队比大型团队具有更低的失败可能性。从这点上,微服务架构还能够降低团队组织级别的风险。

另一方面,软件团队中时常会出现所谓的“扯皮现象”。这是一种明显的过程浪费,该现象的产生原因在于不清晰的边界。微服务架构的一大特征就是根据业务来组建团队,某个特定业务局限于某个微服务中并由独立团队负责所有相关事宜。明确的边界有助于减少团队之间的扯皮现象,从而提升开发效率。



## 2. 快速产品开发

在互联网行业中，时机可能比任何其他因素更为重要。在一些行业中，市场机会窗只会开放很短的一段时间。在这种背景下，产品能不能成功在很大程度上取决于产品投放市场的时间。如果在同等产品规划和运营策略下，也即在相同的业务创新条件下，技术创新就会成为影响产品成功的决定性因素。技术创新能够在改善产品用户体验和缩短产品研发生命周期上提升产品成功的概率，如图 1-13 所示，通过缩短开发时间从而快速推出新产品能带来产品收益上的增长，而对于互联网产品而言，很多时候错过产品发布时机就意味着再也没有机会。

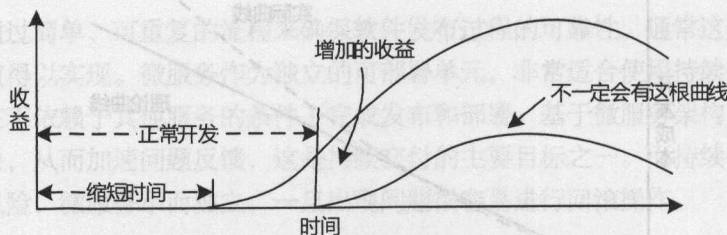


图 1-13 通过技术创新缩短同行业产品开发周期

微服务架构从技术角度给出了缩短产品开发周期的方法，主要表现在并行的开发模式上。将业务拆分成各个微服务能够让不同的业务功能处于一种并行开发的状态，因为每个团队所负责的业务需求只影响到团队自身的微服务，所以各个团队能够独立开发，整个系统也很容易分布到各个团队中。如果涉及简单业务需求的变更或者是发布部署的要求，独立的微服务之间也不需要太多的统一协调工作。大规模的统一协调工作通常只发生在业务结构产生重大变更的场景，而这种场景对于软件开发而言显然是应该尽量避免并进行提前规划。

## 1.5 微服务架构的挑战

将一个系统分散到多个微服务中使得系统整体结构变得更加复杂，这在技术和架构上同样给我们带来了一定挑战。同时，由于各个微服务都需要独立部署，我们就会有更多的服务需要单独进行发布和运行管理，运维基础设施也会面临比单块系统更大的困难。最后，微服务架构在很大程度上改变了系统的研发过程，从团队协作到需求管理等方面，都需要通过引入变化进行调整和完善。

### 1.5.1 技术架构挑战

微服务架构是一种分布式架构，各个微服务之间需要通过网络通信完成交互。网络在连通



各个微服务的同时也会产生响应时间的延迟和通信过程的不可靠性。关于分布式系统固有的问题我们已经在 1.1.2 节中做了介绍，这里不再赘述。另外，我们还需要注意到微服务架构中的去中心化思想和服务版本控制所带来的系统复杂性。

## 1. 去中心化与平衡

微服务架构所带来的独立性可能导致每个服务采用不同的技术体系。去中心化的设计思想意味着微服务之间不需要共享技术，然而缺少通用的技术体系同样也会加剧系统的复杂度。去中心化是微服务架构的基本特征之一，但当每个微服务团队都采用自身团队擅长的不同技术进行微服务构建时，导致的问题就是没有一个人能够明白系统中所采用的所有技术。尽管多数情况下，我们并不需要深入到其他团队的微服务中，但当从统一发布和运维等角度去看待整个系统时，这种技术复杂性就可能会是一个问题。我们需要把控中心化和去中心化之间的平衡性。

## 2. 服务版本控制

每个微服务都应该具备与其他服务保持相互独立的版本控制机制，我们提倡为每个微服务建立版本并根据业务迭代更新这个版本。如果每个服务都能够完全按照自身服务的演进过程进行独立发布，那么事情还比较简单。但如果某一个服务与另一个服务具有版本关联性，即这两个服务要么都不发布、要么一起发布，那么事情就变得有点复杂。如果这些具有版本关联性的服务很多且版本的更新频率很高，如何正确地管理服务版本就是一项具有挑战性的工作。

### 1.5.2 研发过程挑战

在技术之外，采用微服务架构也可能带来研发过程上的挑战。

#### 1. 需求的边界

对研发过程而言，分散的服务在一定程度上等同于分散的业务需求，来源于微服务具有的业务独立性和明确的服务边界。但对于整个系统而言，需求的边界并不会那么容易划分。当我们面对微服务架构，如何确定业务功能的粒度、如何把非功能性需求分解到各个微服务中、如何从系统整体上把握需求的优先级等都是我们不得不面对的问题。

#### 2. 引入变化

对于大多数尚未采用微服务架构的团队而言，使用微服务架构就是一个引入变化的过程。关于团队引入变化，很多人存在一些误解，其中最大的误解就是认为新想法一旦引入就意味着已经成功。当我们尝试采用微服务架构时，团队管理人员想了很多办法、做了很多工作，找到了问题的切入点，跟各个利益方讨论之后终于引入了大家都赞同的变化，然后就期望这个变化



能按照自己想的那样发挥效果，这是不对的。当微服务架构被引入时，我们还要做很多事情，因为前面所提到的各种技术、架构和过程的挑战都需要我们进行跟踪和协调。

## 1.6 实施微服务架构

前面分析了微服务架构给我们带来的很多优势以及实施的挑战，但还是要总结一点，那就是对于很多领域而言，微服务架构是一种改进机制，不同类型的项目和产品都可以从微服务架构转型的过程中获得收益。

### 1.6.1 微服务架构实施前提

图 1-14 来自马丁·福勒（Martin Fowler）的文章<sup>⑥</sup>，揭示了生产率和复杂度的一种关系。在复杂度较小时，采用单块系统的生产率更高，微服务架构反而可能降低生产率。当复杂度到了一定规模时，无论采用单块系统还是微服务架构，都会降低系统的生产率。但是单块系统的生产率开始急剧下降，而微服务架构则能缓解这种生产率下降的程度。图 1-14 展示了复杂度和生产率拐点的存在，但并没有量化复杂度的拐点到底是多少。也就是说，系统或代码库的规模达到具体多大才适合开始进行微服务化的拆分，需要各个团队因地制宜。只有当出现这个拐点时对系统进行微服务化的拆分才是合适的方案，服务的合理拆分是实施微服务架构的一大前提。

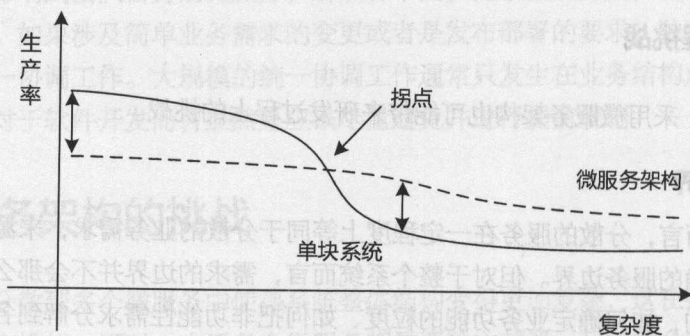


图 1-14 生产率和复杂度关系图

微服务架构实施的另外一个前提是基础设施的自动化。在单块系统中，一个应用部署到一台主机或在主机集群上，部署复杂度是  $O(n)$ 。而当把单块系统拆分成多个微服务之后，其部署复杂度就上升到  $O(n^2)$ 。如果缺乏自动化基础设施，仅实现服务的部署就是一件工作量大、风险高的工作。所以微服务的实施才需要基础设施自动化，这和服务规模有关，从开发之后的构建、测试、部署都需要高度自动化的环境来支撑才能有效降低边际成本。



## 1.6.2 微服务架构实施模式

实施微服务架构存在固定的模式,最典型的实施模式就是从一个单块系统开始逐步转变到多个维度的微服务架构。通常,不同的业务功能根据优先级、工作量等角度转化为一个一个的微服务。促使这种转变的主要目的可能来自于业务的快速变化和迭代要求,或者来自于将系统的部署和运维变得独立而简单,而更为重要的原因则可能来自于对架构可扩展性和可伸缩性的考虑。当然,从单块系统转变到微服务架构,在方式上不同团队、不同场景可能也存在差异性。对于那些对服务的可用性要求很高的场景而言,可以先通过使用一些服务隔离手段,设法提高服务的整体可靠性之后再拆分微服务;而有些场景下,对业务的分离可能是系统的最大痛点,那么快速、合理地拆分业务就是实施微服务架构的第一步。

当然,现实中也存在从无到有实施微服务架构的机会,尽管这种机会非常少见。即使有这种机会,一般的做法也会是先构建一个单块系统,再设法做改进。因为从无到有的过程就是一个从小到大的过程,而正如前文提到的微服务实施前提,小系统没有必要直接采用微服务架构。但在系统架构的初始设计阶段,采用领域驱动的设计思想建立粗粒度的领域模型是一种最佳实践,有助于后续向微服务架构转型的实施过程。领域驱动设计同样也驱使技术团队在设计阶段就考虑到合适的团队分工和各个团队之间的独立性。

混合式是尝试微服务架构的第三种方式,因为微服务架构很容易与现有系统并存,这些微服务能够很好地对系统起到补充作用。如果需要,我们能够很快实现很多微服务,而一旦问题出现,移除这些微服务的成本也较低。微服务与遗留系统之间的易整合性是我们采用微服务的一种主要原因,在本书的后续章节中也会专门介绍这种整合关系以及实施方式。

## 1.7 本章小结

本章作为全书的开篇,围绕微服务架构的定义和特征展开讨论。微服务架构是一种新型的架构设计模式,是对传统单块系统的改进。同时,微服务架构基于分布式系统,但又与分布式系统存在区别。

微服务架构并不是一个全新的事物,而是在现有 SOA、ESB 等架构体系的基础上发展和建立起来的开发模式。微服务也不是一个纯技术的事物,而是涉及业务边界、基础设施、组织架构等的综合体系。微服务架构存在明显的优势,但也面临着各种挑战。

本章还提炼了实施微服务架构的基本方法,包括实施的前提和实施的模式。而服务模型、实现技术、基础设施和研发过程等核心要素构成了构建微服务架构的系统方法,同时也形成了本书的行文框架,为后续内容展开做好铺垫。



## 第二篇 服务建模

### ◆ 本篇内容

本篇共有两章，关注于微服务建模。服务建模是微服务架构得以实施的前置条件，因为服务的本质还是为了更好地实现业务功能。微服务建模包括两个主要组成部分。

1. 服务建模方法。服务建模方法用于明确服务模型的各个维度和表现形式，具体包括服务的分类以及与业务的关系、服务的概念模型及统一的表现形式、基于业务领域进行服务边界划分的方法以及服务数据的建模等切入点。

2. 服务拆分和集成。通过服务拆分，我们将单块系统分解成微服务体系，而通过服务集成，再把各个微服务构成一个整体，从而完成具体业务流程。服务拆分有固有的维度，也需要采用一定的策略。通常，我们会侧重于管理服务的依赖管理、数据和事务边界。服务集成包括RPC、REST、消息传递、服务总线、数据复制、客户端集成和外部集成等技术实现方案。



## 第2章

# 服务建模方法

如同系统架构设计首先需要进行业务建模一样，微服务架构的设计同样从服务建模开始。一般而言，软件行业建模工作的目的在于将业务需求转变为一种能够直接用于软件实现的载体，以 UML（Unified Modeling Language，统一建模语言）为代表的建模语言是这种载体的具体表现形式。在微服务架构中，服务建模的目的同样是为了便于开发人员开展后续的服务设计和实现工作。在本书中，微服务建模的切入点有如下四个方面。

- 服务分类

从技术实现角度和业务角度分别切入，梳理微服务架构中的代表性服务类型和表现层次。

- 服务模型

提供服务的概念模型，并给出服务的统一表现形式。

- 服务边界

服务边界是服务建模的核心要素，采用面向领域思想，通过识别业务领域边界并确定界限上下文来达到明确服务边界的效果。

- 服务数据

对于微服务而言，传统的规范化数据模型存在的问题，需要通过数据去中心化手段实现对服务数据的有效管理。

本章将对以上各个方面详细展开讨论，以便读者能够从建模角度理解微服务的设计思想。

## 2.1 服务分类

对于服务而言，一方面可以从技术实现角度进行归类；另一方面，服务也具备层次性，需要把服务与业务结合起来梳理服务层次。本节将分别从以上两个方面探讨服务的系统分类方法并给出相关示例。



## 2.1.1 服务的基本类别

技术平台在构成过程中可以采用大量成熟的技术理念和工具，基本思路就是实现服务化。一般认为服务有工具服务、实体服务和任务服务等三种主要的表现类型。

### 1. 工具服务

工具服务（Utility Service）代表可重用服务，区别业务模型。作为应用程序与技术基础设施之间的交叉点，工具服务的特点是业务领域无关，本质是面向技术、具备高可重用性的底层处理服务，因此能够遵循独立开发和管理生命周期。

工具服务的识别遵循如图 2-1 所示的模式，以 Java 语言为例，工具服务的识别包括 Java 标准的 API 的封装、公共功能区域的提炼、非功能性需求的抽取以及常见开源框架的应用等四个维度，图 2-1 展示了提取工具服务的这四个常见维度。因为其业务无关性，相对于其他两种服务，工具服务相对比较容易提炼。

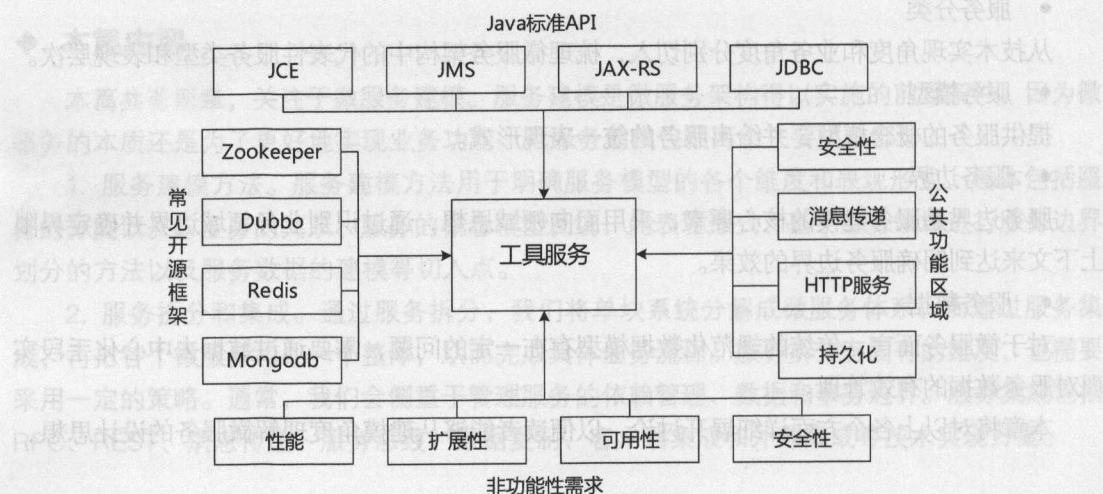


图 2-1 工具服务的识别

Java 领域中存在很多开放、强大而常用的 API，如与安全性相关的 JCE（Java Cryptography Extension，Java 密码扩展）提供用于加密、密钥生成和协商以及消息认证码（Message Authentication Code，MAC）算法的框架和实现，JMS（Java Messaging Service，Java 消息服务）提供面向消息中间件（Message Oriented Middleware，MOM）的 API 规范，JAX-RS（Java API for RESTful Web Services）则支持按照 REST 风格创建 Web 服务。这些 API 可以分别对应到公共功能区域中的安全性、消息传递系统、HTTP 数据



传输等各个领域，构成工具服务中的一大类型。同时，围绕系统的性能、可扩展性、可用性等因素，工具服务也包含了基于目前主流开源框架的封装需求，如使用 Zookeeper 实现分布式协调和分布式锁机制，使用 Dubbo 实现通用的 RPC 功能和服务治理，使用 Redis 等 Nosql 技术实现海量数据存储等，这些工具的应用都可以独立于业务并形成统一化、开放式服务。

工具服务根据其用途也可以做进一步细分，一般包括以下表现形式。

- 公共工具服务

公共工具服务面向多种应用程序，如安全性、记录和审核，该类工具服务通常设计成基于 Web 的服务，并开放通用、松散类型接口。

- 资源工具服务

资源工具服务封装物理系统资源，如数据存储/消息资源，这类服务处于底层，使用服务门面暴露入口。

- 微工具服务

微工具服务细粒度、高度特性化，如 XML 加密服务。这类服务通常本地调用，需要考虑性能、无状态性和线程安全，可以作为 JAR 包进行直接引用。

- 包装器工具服务

包装器工具服务，面向遗留系统，建立标准化服务契约，显然这类服务需要明确所支持的数据和消息模型。

## 2. 实体服务

实体服务（Entity Service）建立一种一致的方法访问和处理业务数据，对应基于业务的功能上下文，侧重于以数据为中心。实体服务同样可重用，一般被更高级的任务服务使用。实体服务的层次如图 2-2 所示，在业务流程中包括各种遗留系统、数据库和外部系统在内所组成的数据孤岛之间形成一种实体服务，其本质是提供一种规范化的数据模型。

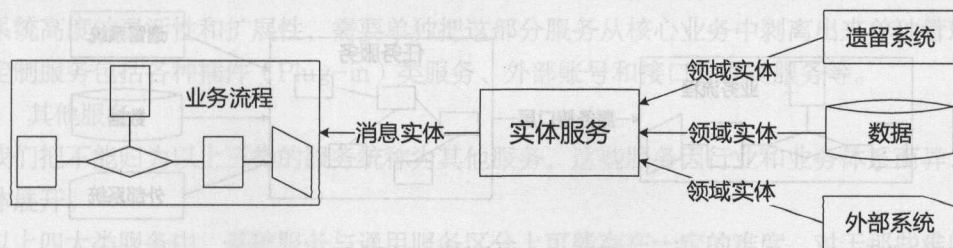


图 2-2 实体服务的层次

实体服务中包括领域实体（Domain Entity）和消息实体（Message Entity）两种实体。领域实体是指服务中规范化的业务数据模型，而消息实体特定于实体服务契约的数据模型，这



两种实体体现在图 2-2 中就是实体服务的左右两端。对于常见的用户（User）这个概念而言，领域实体中可以包括账户（Account）、订单（Order）和联系方式（ContactInfo）等信息。但当实体服务暴露对外的接口时，可能并不一定需要所有的领域数据，消息实体作为领域实体的包装器和简单版本，就能成为一种灵活的、与领域实体相互独立的数据传输和转换机制，领域实体和消息实体示例如图 2-3 所示。

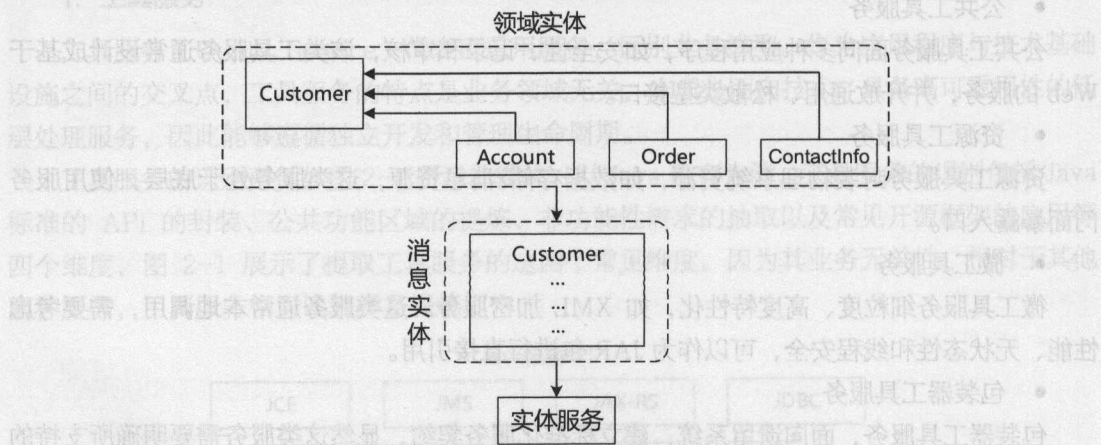


图 2-3 领域实体和消息实体示例

### 3. 任务服务

任务服务（Task Service）关注实现业务相关逻辑，很大程度上由组合逻辑组成，通常需要维护状态。任务服务在靠近组合服务的位置承载，体现为一种组合结构（见图 2-4）。在任务服务中，确保使用实体服务限制返回数据量，服务调用携带消费者信息和上下文，并由任务服务决定事务边界。

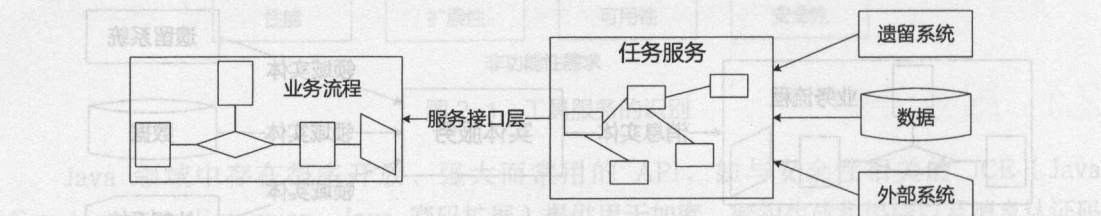


图 2-4 任务服务的组合

以上三种服务化思想都可以用于构建微服务，工具服务因为不涉及业务，识别和提取相较于另外两种服务而言比较简单。实体服务关注于数据，任务服务关注于业务组合，都需要根据业务本身抽象化。



## 2.1.2 服务与业务

服务建模本质上是一个为了满足业务需求，并通过技术手段将这些业务需求转换为可实现服务的过程。从业务维度，我们同样可以对服务做层次分类。

### 1. 业务服务的层次分类

服务围绕业务能力建模，而业务能力体现的是一种分层结构，因为不同业务体系在整个产品线或产品平台中的定位往往有所不同。一般而言，我们可以把业务体系中的服务分成如下几种类型。

#### • 基础服务

基础服务处于业务体系的最低层，这些服务一般对用户并不可见，但却被其他各种服务所依赖，或者说为其他服务提供运行时支撑。比较典型的基础服务包括消息服务、路由服务等。从命名上，我们可以看到这些服务是业务体系中与技术结合比较密切的一组服务，有时候也很难区分基础服务中业务和技术之间的隔离点，因为基础服务往往与技术整合在一起。

#### • 通用服务

通用服务与基础服务的区别就在于它们是完全面向业务的服务，而且这些服务通用性非常高，也可以像基础服务一样提供低层的业务支撑。取决于对业务的抽象程度，通用服务包含内容可以很多，如账号服务、登录服务、通知服务等对于任何一个系统而言都可以认为是一种通用服务。这是微服务架构中最需要也最应该建模的一组服务。

#### • 定制服务

在一个系统中，定制服务相对而言不应该太多，而且在日常开发过程中，我们会发现定制服务面向外部、面向系统集成类的表现形式比较多。一方面，面向外部、面向系统集成类的业务需求需要对第三方服务进行适配，通常都需要进行一定程度的定制化开发；另一方面，为了确保系统高度的灵活性和扩展性，需要单独把这部分服务从核心业务中剥离出来单独管理。常见的定制服务包括各种插件（Plug-in）类服务、外部账号和接口管理类服务等。

#### • 其他服务

我们把不能归为以上三类的服务统称为其他服务，这些服务因行业和业务体系而异，这里不具体展开。

以上四大类服务中，基础服务与通用服务区分上可能存在一定的难度。对于那些难以区分的服务而言，我们也可以从服务被依赖的数量出发做一个简单判断：如果一个服务被依赖的层次和数量非常多，那么可能会更加偏向于基础服务，反之则可以归为通用服务。



## 2. 业务服务示例

根据业务服务的层次分类，我们给出一个具体服务分类的示例（见图 2-5）。该示例来源于移动医疗行业场景。移动医疗系统是互联网+背景下传统医疗业务与互联网技术的碰撞，用于帮助患者解决挂号、就诊、支付、查看检查检验报告等场景中的痛点，将部分线下业务转移到线上，从而提供更好的用户就医体验。关于该示例，我们在本书后续章节还会具体展开讨论。

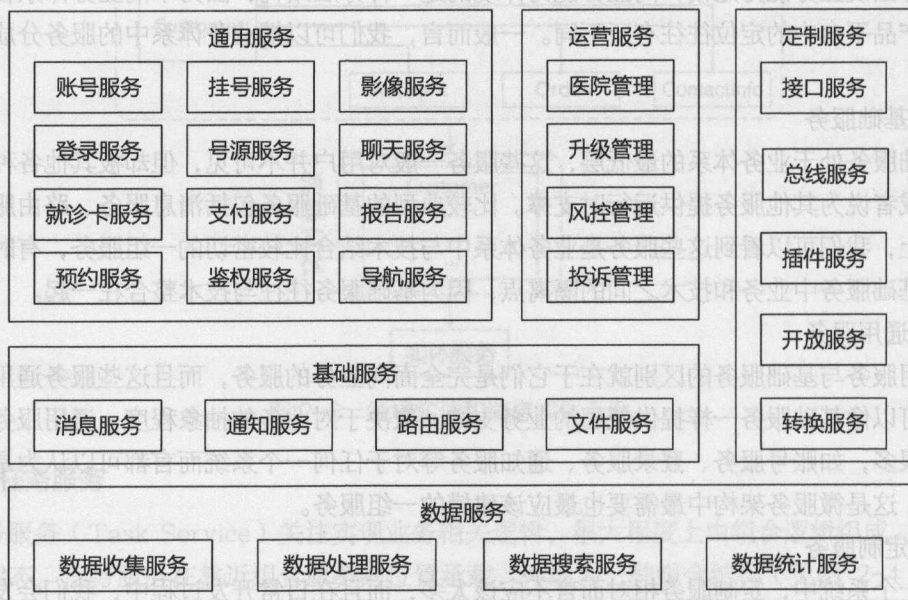


图 2-5 移动医疗系统服务架构图

从图 2-5 中，我们可以看到，对于一个典型的移动医疗系统而言，基础服务可以包括消息服务、文件服务等；而通用服务内容广泛，图中仅列出了一部分，包括账号服务、登录服务、就诊卡服务、搜索服务、预约挂号服务、检查检验报告查看服务、影像服务、支付服务等；而涉及与医院或第三方系统进行对接的服务都可以归为定制化服务，包括接口服务、数据转换服务等。图 2-5 还展示了运营服务和数据服务等具体产品形态和行业的其他服务。

## 2.2 服务模型

架构师使用模型来表达系统架构，对于微服务系统而言，我们也需要一种特定模型来描述服务本身。



### 2.2.1 服务的概念模型

服务的概念模型来自于两个维度：一个是服务标准，即什么样的服务才是一个好的服务；另一个是服务级别，即不同的服务应该具备不同的重要性。

#### 1. 服务标准

关于服务标准，SOA 体系中的很多设计理念同样适合于微服务架构。以下设计原则就是对一个服务标准最好的描述。

- 服务无状态

服务无状态（Service Statelessness）是指服务通过推迟或避免状态信息的管理，从而最小化资源消耗。具备无状态的服务通常有一些明显的设计特征。例如，高度业务流程无关的逻辑，使得服务没有被设计为保存任何特定业务流程中的状态信息；服务契约的约束很少，从而能够在运行时接收和传输更广泛的状态数据等。服务无状态性有助于增强服务的可扩展性。

- 服务可重用

要实现服务可重用（Service Reusability），首先需要确保建立无关功能性的上下文（Context）结构，也就是说与服务封装在一起的上下文对任何使用场景都有足够的无关性，这样服务才能被认为具备可重用性。同时，服务内部的业务逻辑足够通用，以便能够用到不同类型的服务消费者的众多场景中。而且，服务逻辑可以被并发访问，服务设计为在一个或多个消费者同时访问时具备同样的访问效果。

- 服务可发现

服务可发现（Service Discoverability）是指服务具备能够用于传递的元数据构建能力，通过这些元数据可以有效地发现和解释服务。如果存在服务注册中心，那么可以通过这些元数据与注册中心之间建立服务注册和发现机制。如果没有注册中心，我们也需要通过合适的服务描述语言，构成服务提供者与消费者之间的服务约定。

- 服务自治

服务自治（Service Autonomy）是指服务对其低层运行时环境具有高度的控制权。显然，为了实现服务自治，服务契约应该表达定义明确的功能边界，这个边界不应该与其他服务的功能边界相重叠。同时，服务应该被部署在一个独立而隔离的环境中，承载服务的这个环境也应该具备能够处理高并发的访问能力，以便更好地实现服务可伸缩性目的。

- 服务松耦合

服务松耦合（Service Loose Coupling）的主要目的在于为消费者提供较低的耦合度要求，通常表现在服务提供者和服务消费者能够以适应性的方式随时间进行自我演化，彼此之间的影响达到最小。在实现服务的过程中，服务松耦合原则强调服务契约与技术实现细节上的解耦。



关于这些原则的进一步描述，可以参考相关资料<sup>[6]</sup>。

## 2. 服务级别

可以从发生具体事故时服务对用户体验的影响、所造成的经济损失等角度对服务进行具体分级。在服务分级上，一种常见的分级方法是将系统服务分成三个不同的等级。

- 一级服务

一级服务具备完善的容错降级机制及对低级别服务的熔断措施、定期压测、配置高级别的监控告警流程等。

- 二级服务

二级服务多采用异步方式进行系统交互，容忍暂时数据不一致性。

- 三级系统

三级服务则可随时降级整个服务。

### 2.2.2 服务的统一表现形式

服务需要具备统一的表现形式，也就是需要具备契约化的约束条件，而这种契约化的约束条件一般可以通过文档的方式进行展现。

#### 1. 服务契约化

对于每一个服务而言都应该提供了一份契约文档，并发布到统一的管理平台以便服务相关人员查看和更新。服务契约（Service Contract）化要求至少对服务的基本方面做出说明，包括以下几项。

- API

具体接口的 API 接入技术说明。

- 能力

服务能力的描述，包括服务所属的业务模块，所能提供的业务功能以及具体的应用场景。

- 约束

提供这些能力所约定的一些限制条件说明，这些限制条件多数与业务场景有关。现实中，以下这种情况并不少见，即同一个接口定义以及其所代表的业务能力在不同的场景中会出现不同的表现形式。

- 版本

支持的最新和历史的版本说明。这一点对于微服务而言是必备的要素。

显然，将服务契约化有助于在开发人员之间形成统一语言，减少多余且可能反复的口头沟通，降低协作成本。同时，已经形成文档化的服务契约是一项重要的组织过程资产



(Organizational Process Assets), 对于系统的维护和管理同样意义重大。

## 2. 文档服务

Swagger (<https://swagger.io/>) 应该是目前最流行的 API 描述工具。通过 Swagger, 微服务的开发人员可以在代码中简单添加几个注解用于描述 API, 然后产生一个友好的 Web 界面供使用者获取 API 定义相关的参数类型、参数名称等各项信息 (见图 2-6)。同时, 它还允许在这个界面上直接执行请求并获取请求的对应结果, 这在前后端开发联调阶段非常有用。

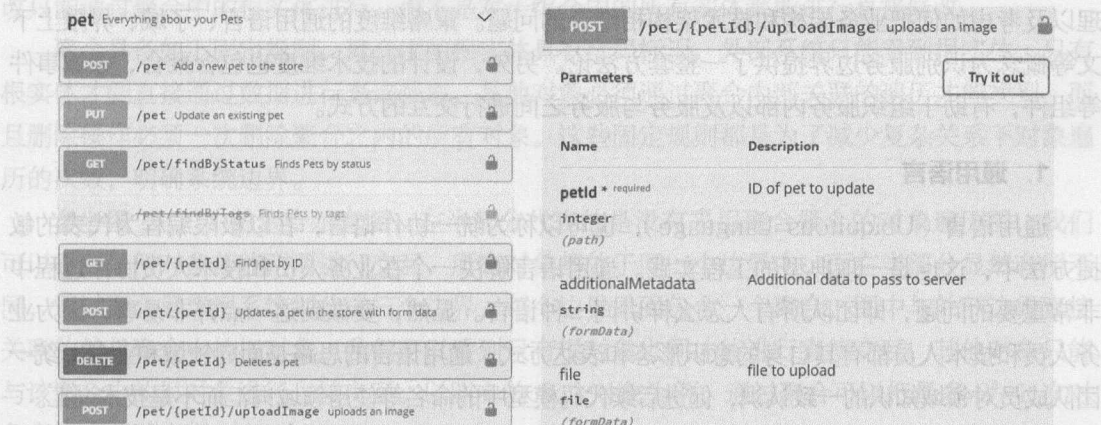


图 2-6 Swagger API 定义效果图 (来自 Swagger 官网)

还有一个可以采用的文档服务是 HAL (Hypertext Application Language, 超文本应用程序语言), 我们将在下一章做相关介绍。

## 2.3 服务边界

在明确了服务分类和服务模型之后, 接下去的工作就是识别服务, 识别服务的切入点在于识别服务与服务之间的边界 (Boundary)。明确服务边界是服务拆分和集成的前提, 关于服务拆分和集成的具体实现方式我们将在下一章专门介绍, 本节内容将围绕如何识别和划分服务边界展开讨论。

### 2.3.1 识别业务领域及边界

在微服务架构中, 识别服务边界的方法主要参考领域驱动设计 (Domain-Driven Design, DDD) 思想<sup>[7]</sup>。所谓领域 (Domain), 即是对现实世界问题的一种统称, 是一个组织



的业务开展方式，体现一个组织所做的事情以及其中所包含的一切业务范围和所进行的活动，我们在开发软件时面对的就是组织的领域。例如，一个电商网站的领域包含了产品名录、订单、库存和物流的概念，而医疗信息化公司关注挂号、就诊、用药、健康报告等领域。可以简单认为，我们所要设计的每一个特定的业务系统就是一个领域。

在领域驱动设计中，有两个主要的设计维度，即设计的策略维度和设计的技术维度。其中设计的策略维度关注如何设计领域模型以及对领域模型的划分，其目的在于清楚界分不同的系统与业务关注点。策略维度是一个面向业务、具备较高层次的设计维度，偏重于业务架构的梳理以及考虑如何把业务架构和技术架构相结合的问题。策略维度的通用语言、子域、界限上下文等概念为识别服务边界提供了一整套方法论。另外，设计的技术维度也包含聚合、领域事件等组件，有助于组织服务内部以及服务与服务之间进行交互的方式。

## 1. 通用语言

通用语言（Ubiquitous Language），也可以称为统一协作语言，在以极限编程为代表的敏捷方法中，这也是一项典型的工程实践。通用语言解决一个在业务人员和技术人员协作过程中非常重要的问题，即团队所有人怎么样讲同一种语言。显然，要做到这一点并非易事，因为业务人员和技术人员都有其自身的意识形态和表达方式。通用语言的思路是面向领域和业务，统一团队成员对领域知识的一致认识，促进后续代码模型中的命名等使用领域词汇而不是技术词汇。

通用语言的好处在于能够将客户、体验设计师、业务分析师、技术人员集结在一起对业务需求进行沟通，随后对其进行领域划分。这是服务能够有效识别的前提。

## 2. 子域

针对如何进行系统划分，领域驱动设计给出了子域（Sub-Domain）的概念。子域作为系统拆分的切入点，其来源往往取决于系统的特征和拆分的需求，如核心功能、辅助性功能、第三方功能等。

子域的划分虽然因系统而异，但通过对子域的抽象，我们还是可以梳理出通用的分类方法。业界比较认可的分类方法认为，系统中的各个子域可以分成核心子域、支撑子域和通用子域三种类型，其中系统中的核心业务属于核心子域，专注于业务某一方面的子域称为支撑子域，可以用于整个业务系统且作为一种基础设施的功能可以归到通用子域。当然，我们可以根据需要建立对子域的抽象模型，为了描述方便，本节后续内容以上述的分类方法标记系统子域。

关于如何区分子域的三种类型，并没有一个统一的标准。正如前面提到的领域的概念，业务流程代表了一个业务领域，业务流程所涉及的数据和角色或是通用子域，或是支撑子域，由其在这个业务流程中的角色和竞争力所决定。以统一身份认证功能为例，一般本身并不产生业务价值，不是业务成功的促成因素，但是为所有流程提供了入口，可以归为通用子域。而同样



的统一身份认证功能，在一个专门提供身份认证机制的业务流程中，显然就是一个核心子域。

### 3. 聚合

一个子域中包含若干聚合（Aggregate）。聚合的核心思想在于将关联减至最少有助于简化对象之间的遍历，使用一个抽象来封装模型中的引用。聚合的组成有两部分，一部分被称为根（Root）实体，是聚合中的某一个特定实体；另一部分描述一个边界，定义聚合内部都有什么。聚合代表一组相关对象的组合，是数据修改的最小单元，也就是意味着对对象组合的修改只能通过聚合中的根实体进行，而不是对于组合中的所有实体都能进行直接修改。

聚合具备如下固定规则。聚合中的根实体具有全局标识、外部系统只能看到根实体，只有根实体才能直接通过数据进行查询获取，其他对象必须通过聚合内部关联的遍历才能找到，而且删除操作必须一次删除聚合之内的所有对象。这些固定规则都是为了减少复杂关系下对象遍历的次数，明确系统边界。

参考图 2-7 中的聚合示意图，左半部分代表的是没有采用聚合概念的对象遍历图，我们可以看到任何对象都能两两进行交互，所有对象都处于同一个边界中；而右半部分显然有所不同，通过聚合思想把系统划分成三个边界，每个边界里面包含一个聚合，图中与外部边界直接关联的就是聚合中的根对象，我们可以看到只有根对象之间才能进行直接交互，其他对象只能与该聚合中的根对象进行直接交互。以图中的 8 个对象为例，通过聚合可以把最多  $2^8-1$  次对象直接交互减少到  $2^3-1$  次。

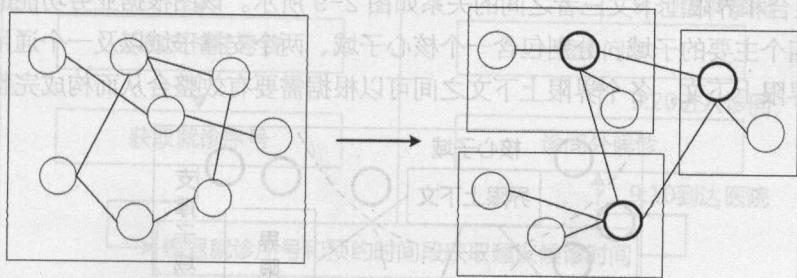


图 2-7 聚合示意图

聚合被证明是开发微服务的关键，使用聚合将领域模型分散和参与到每个聚合中，这也使得领域模型更容易理解。同时，聚合也能帮助我们明确查询操作和删除操作的范围。

同时，因为对象的访问只能通过聚合中的根实体进行，考虑到性能和伸缩性，我们一般建议聚合的大小不应该过大。



## 2.3.2 界限上下文

界限上下文（Boundary Context）顾名思义就是根据界限确定的一个上下文环境。在软件开发领域，上下文代表着具体的业务场景，而界限为各个上下文之间提供边界。

### 1. 领域与界限上下文

#### （1）领域与界限上下文概念

子域存在于界限上下文中，这里的界限指的是每个模型概念、属性和操作，在特定边界之内具有特定的含义，这些含义只限于该界限之内。图 2-8 就是一个简单的界限上下文，其中 A 上下文和 B 上下文中都存在 User 对象，但是 B 上下文中的 User 对象不同于 A 上下文中的 User 对象，而 B 上下文中 Account 对象可能基于 A 上下文中的 Role 对象，这时候我们就会发现界限的划分能在很大程度上影响系统的设计和实现，也为确定服务边界提供了合理性依据。

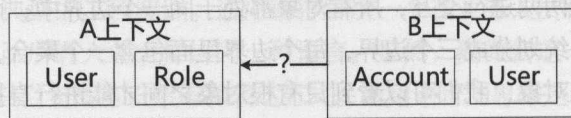


图 2-8 界限上下文示例

子域、聚合和界限上下文三者之间的关系如图 2-9 所示。该图根据业务功能的特性把整个系统拆分成四个主要的子域，分别包含一个核心子域、两个支撑子域以及一个通用子域，每个子域都有其界限上下文，各个界限上下文之间可以根据需要有效整合从而构成完整的领域。

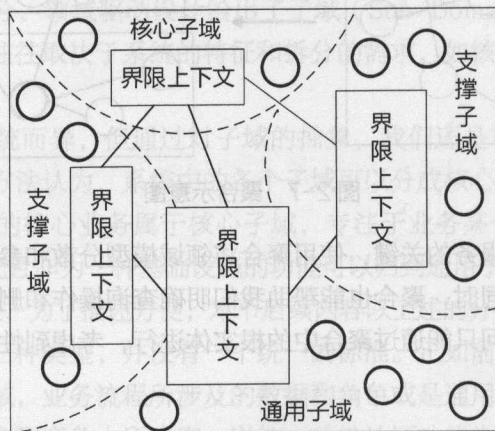


图 2-9 子域、聚合和界限上下文三者之间的关系



从图 2-9 中我们也可以进一步看到, 聚合位于子域中, 与子域一样代表的也是一种拆分的层次和粒度。通过在子域的基础上再添加一层聚合的抽象, 使得系统拆分不仅仅在大的业务体系级别有所体现, 也精确到了系统内部的对象级别。聚合概念的提出与软件复杂度有直接关联。软件设计中的一大问题就在于大多数业务系统中的对象都具有十分复杂的联系, 现实世界很少有清晰的边界。复杂的关系需要通过关联数量庞大的对象才能建立, 复杂关系的开发和维护需要投入成本, 也是我们为什么要推行微服务架构的一个根源。

## (2) 子域和界限上下文示例

我们再回顾一下 2.1.2 节提到过的移动医疗系统。移动医疗系统中门诊服务是业务承载的重点。门诊就医服务的主要业务流程如图 2-10 所示。

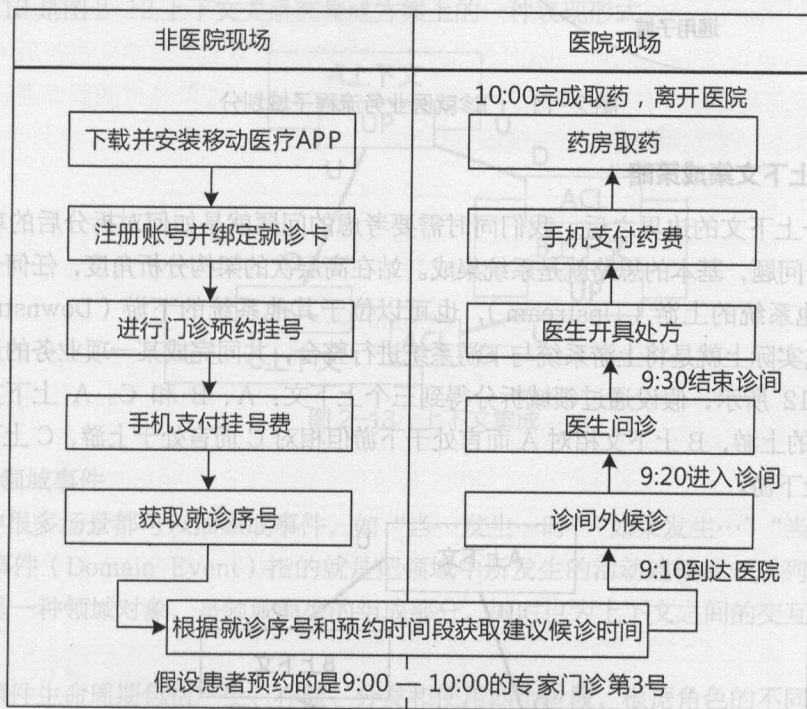


图 2-10 门诊就医业务流程

根据图 2-10 中的整体业务流程, 我们可以分别梳理对应的核心子域、通用子域和支撑子域 (见图 2-11)。



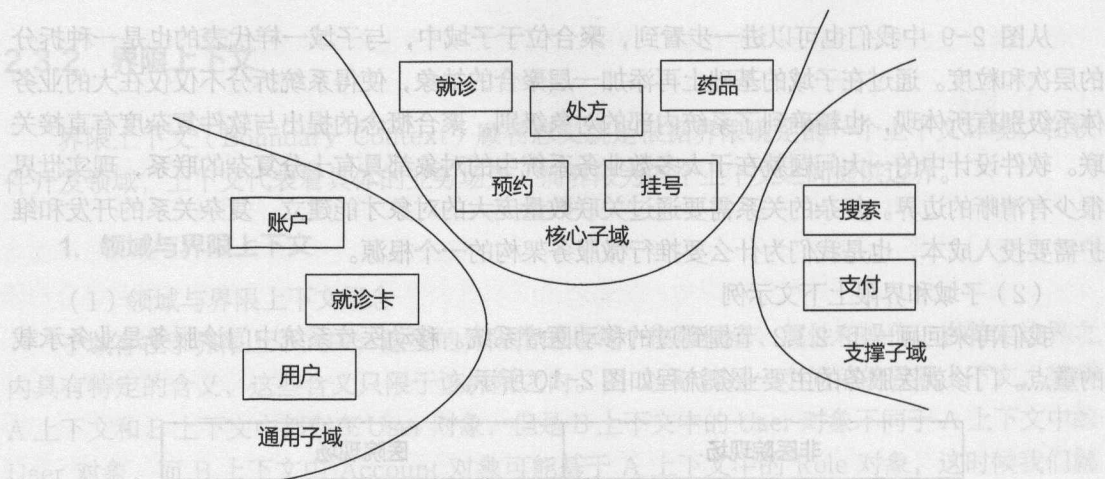


图 2-11 门诊就医业务流程子域划分

## 2. 界限上下文集成策略

明确各个上下文的边界之后，我们同时需要考虑的问题就是如何对拆分后的功能进行组装。对于这个问题，基本的思路就是系统集成。站在高层次的架构分析角度，任何一个系统都可以处在其他系统的上游（Upstream），也可以位于其他系统的下游（Downstream）。所以，系统集成实际上就是将上游系统与下游系统进行整合，共同完成某一项业务的过程。

如图 2-12 所示，假设通过领域拆分得到三个上下文：A、B 和 C。A 上下文同时位于 B、C 上下文的上游，B 上下文相对 A 而言处于下游但相对 C 而言处于上游，C 上下文则处在整个系统的最下游。

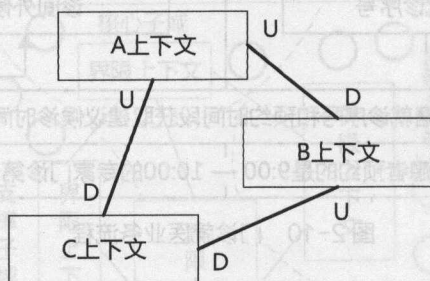


图 2-12 上下文关系示例

上下文集成的基本思路在于解耦和统一。对于解耦而言，一方面在于技术实现上的依赖性，需要支持异构系统的有效交互；另一方面也需要把关注于集成的实现与业务逻辑的实现相分离，确保集成机制的独立性。而统一的含义在于一致性，即上游系统应该定义协



### (1) 防腐层与统一协议

在对任何子域和上下文进行提取时, 确保从组织关系和集成模式上对上下文集成进行抽象。图 2-13 是图 2-12 上下文关系在集成方案上的一种表现形式。

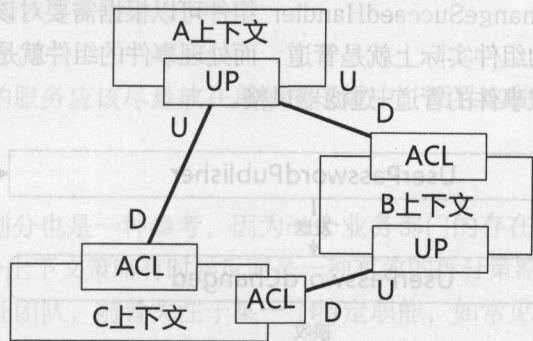


图 2-13 上下文集成

现实中很多场景都可以抽象成事件，如“当…发生…时”“如果发生…”“当…时通知我”等，领域事件（Domain Event）指的就是把领域中所发生的活动建模成一系列离散事件。领域事件也是一种领域对象，是领域模型的组成部分，同时也为上下文之间的交互提供了另一种方式。

事件的识别有时候具有一定的隐秘性，当一个实体依赖于另外一个实体，但两者之间并不希望产生强耦合而又需要保证两者之间的一致性时，我们通常就可以提取事件，这是事件最容易识别的场景。如在移动医疗系统的挂号（Registration）场景，为了避免挂号系统与其他系统之间产生强耦合，当一个挂号结束时，该上下文更新挂号结果信息并通知相关兴趣方，这个过程中我们就可以提取 RegistrationFinished 事件。



领域事件同样需要建模,一般使用过去时对事件进行命名,如上述的 `RegistrationFinished` 事件。领域事件包含唯一标识、产生时间、事件来源等元数据,也可以根据需要包含任何业务数据。同时,领域事件具有严格意义上的不变性,任何场合都不可能对事件本身做任何修改,因为事件代表的是一种瞬时状态。

事件驱动架构的发布-订阅机制非常适合与其他风格进行整合构成复合型架构风格,最典型的就是与管道-过滤器(Pipe-Filter)风格进行整合。管道中流转的数据就是领域事件,而过滤器可以是一个子域中的某个组件,也可以是进行跨子域的界限上下文。如图 2-14 所示就是一个典型的管道-过滤器示例,我们看到事件发布者 `UserPasswordPublisher` 发布了一个 `UserPasswordChanged` 事件,而订阅该事件的 `UserPasswordHandler` 组件对该事件进行处理之后再次发送一个 `UserPasswordChangeSucceed` 事件,负责接收 `UserPasswordChangeSucceed` 事件的 `UserPasswordChangeSucceedHandler` 组件可以根据需要对该事件进行后续处理。整个过程中能够发送事件的组件实际上就是管道,而处理事件的组件就是过滤器,通过管道和过滤器的组合形成基于领域事件的管道-过滤器风格。

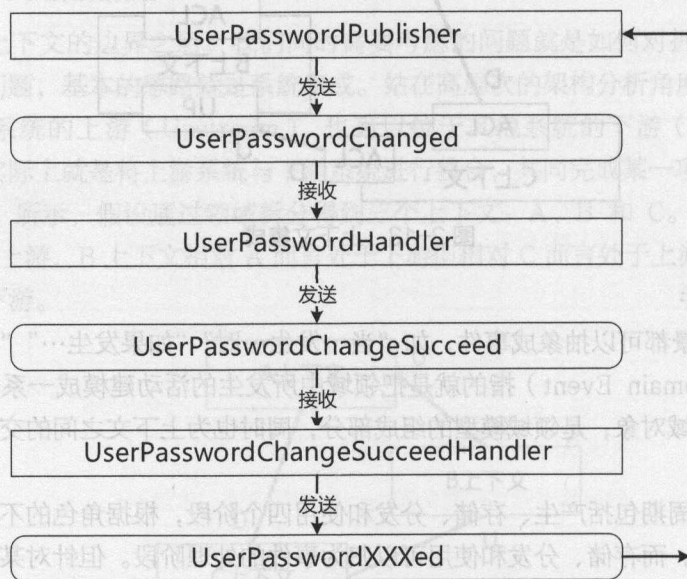


图 2-14 事件与管道-过滤器组合示意图

以上对界限上下文之间的集成策略做了探讨并提供了基本思路,至于具体实现这些集成策略的工具和方法我们将在下一章具体展开。



### 2.3.3 服务边界划分的原则

在通过使用领域和界限上下文进行服务边界划分的过程中，经常会碰到这样一个问题，即这个服务看起来放在这个子域合适，放在另一个子域也合适，如何抉择呢？这就需要梳理服务边界划分的原则。常见的边界划分原则如下。

- 服务关联度原则

该原则有几种表现形式，比如是否该服务变化时，其他服务也需要进行变化；或者说该服务中的数据是否通常在当前上下文中的范围内使用。

- 业务能力职责单一原则

服务边界内的业务能力职责应单一，不是完成同一业务能力的服务不应该放在同一个上下文中。

- 读写分离原则

对于数据读取类型的服务应该尽量放在单独的子域中，而且这种子域一般不应该是核心子域。

- 组织关系原则

组织中业务结构的划分也是一种参考，因为一个业务部门的存在往往有其独特的业务价值。所以，一个团队一个上下文策略有时候反而是一种有效的拆分策略。团队的构建方式可以是职能团队也可以是特征团队，前者关注于某一个特定职能，如常见的服务端、前端、数据库、UI 等功能团队，而后者则代表一种跨职能的团队构建方式，团队中包括各种职能角色。上下文的构建以及界限的划分是一项跨职能的活动，如果团队组织架构具备跨职能特性，可以安排特定的团队负责特定的上下文并统一管理该上下文对应的界限。

在服务边界划分中，应该尽早识别和剥离通用领域（如账户管理、登录等服务），而对于系统中最复杂且相对多变的子域，需要及早进行隔离并充分考虑它与核心子域之间的协作关系。

## 2.4 服务数据

通过确定服务边界，服务在逻辑上就变成了一个个独立的个体，但是我们还要小心一个并不容易进行独立处理的因素，那就是数据。绝大多数服务都会依赖数据，而很多数据可能也会被一批服务所依赖。本节我们将讨论服务与数据之间的关系并给出相关的设计思路。

### 2.4.1 规范化数据模型的问题

规范化数据模型是传统关系型数据库设计的核心，即通过三大范式实现数据的有效存储，



并为开发人员提供一整套对数据的操作方式。规范化数据模型有利有弊，它为如何管理关系型数据提供了最佳设计理念，但同时也限制了数据查询的灵活性和高效性，特别是当查询涉及很多关联（JOIN）场景时，会导致查询性能严重低下。

规范化数据模型的另一个问题在于中心化思想，即把数据统一存储在一个中央数据库中。当大量数据存在于同一数据库时会容易造成数据库访问瓶颈，从而影响数据访问性能，并为系统可用性埋下隐患。

针对规范化数据模型存在的以上问题，微服务架构中数据管理的基本思路就是数据去中心化。

## 2.4.2 数据去中心化

数据去中心化过程也就是数据拆分的过程，在本节中我们将讨论数据去中心化的场景以及对应的处理流程。

### 1. 数据去中心化场景

对于数据拆分需求存在如下几种不同的表现形式。

#### （1）跨表查询场景

跨表查询场景相对比较简单，通常表现为一些数据库表的连接查询操作，不同业务之间共用几张数据库表，而这些表都位于同一个数据库中。应对表连接查询的基本手段就是把数据层连接查询转变到业务层连接查询，也就是说在数据层去掉各种连接操作，将连接查询转变为简单的单表查询，然后对各种表查询的结果在内存中进行动态组装，从而形成业务所需要的数据。跨表查询解决方案如图 2-15 所示。

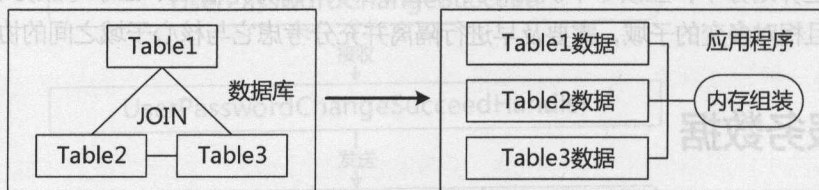


图 2-15 数据层连接转变到业务层连接示意图

从图 2-15 中我们也不难得出结论，如果考虑到未来服务拆分或重构的需要，从一开始就保持数据单表查询是一项最佳实践。

#### （2）跨库查询场景

跨库查询场景相比跨表查询场景就显得比较麻烦，表现在不同数据库之间的表也存在着连接查询操作。这种情况相对较少，但一旦发生所带来的不仅仅是可扩展性问题，也会存在比较



明显的可伸缩性问题。

针对该场景，基本的解决思路有两种。针对一些修改频率不高、相对静态的数据而言，可以采取数据复制的方式达到同一份数据在两个数据库中同时存在的效果，从而将跨库查询转变为同一库中的表查询。数据复制的实现可以简单采用定时同步机制，带有定时数据同步机制的静态数据跨库查询处理方案如图 2-16 所示。

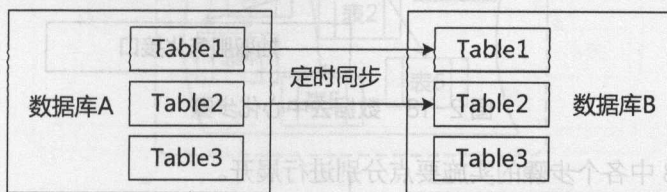


图 2-16 静态数据的跨库查询处理方案示意图

对于那些实时性要求比较高的数据，则可以通过开放接口的方式实现两个数据库之间的数据集成效果，该场景下的解决方案如图 2-17 所示。

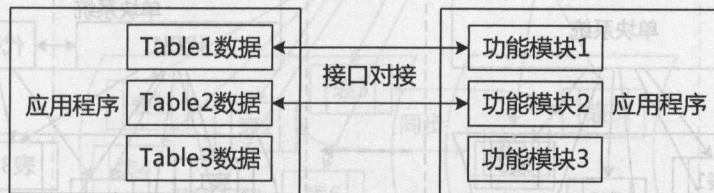


图 2-17 动态数据的跨库查询处理方案示意图

### （3）技术耦合场景

技术耦合场景表现在使用了特定某种数据存储容器相关的技术体系。对于关系型数据库而言，存储过程、触发器等就是典型的数据库工具级别的特有技术。由于每种数据存储容器对这些具体技术的实现和支持方式都有所不同，所以在原则上我们的业务功能都不应该使用这些技术进行数据处理。如果有，那就坚决去掉。采用的方式也是将这些存储过程、触发器中所涉及的业务逻辑全部用代码重新实现一遍即可。

在技术耦合场景中，同样会碰到跨表查询和跨库查询场景，可以综合应用前文介绍的各种方法对其进行重构。

## 2. 数据去中心化流程

在对数据去中心化进行不断尝试的过程中，我们可以得出如图 2-18 所示的步骤。这些步骤一部分参考了《数据库重构》<sup>[8]</sup>一书中提到的方法，同时也是对实践的一种总结。



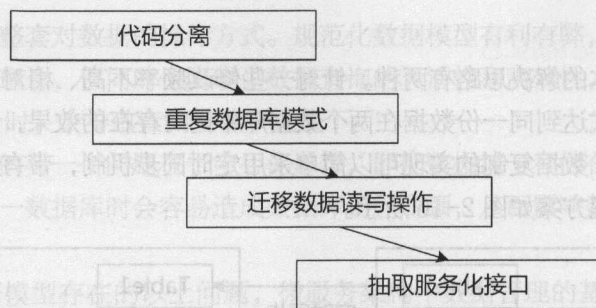


图 2-18 数据去中心化步骤

我们对图 2-18 中各个步骤的实施要点分别进行展开。

### （1）代码分离

要想进行数据库拆分，首当其冲的是代码拆分。这一步相对比较简单，通过物理上的代码拆分，即把原本在单块系统中的代码拆分成几个组成部分就能实现（见图 2-19）。

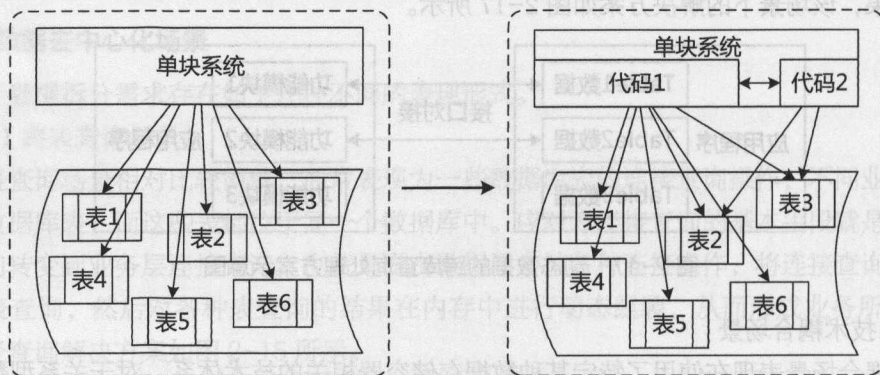


图 2-19 代码分离示意图

### （2）重复数据库模式

代码拆分完之后，对数据的访问会形成两种模式。一种是数据能够随着代码完成拆分，即这些数据不存在多个系统共同访问的情况，那简单把数据迁移出去供单个系统访问即可。但是很多情况下，代码中充斥着跨表查询、跨库查询和技术耦合场景，不能简单进行数据拆分，所以作为一个过渡环节，我们可以把几个系统公用的数据进行冗余处理。冗余处理就是同一份数据库模式和相应的数据同时保存在不同的数据库中。应用重复数据库模式的效果如图 2-20 所示。



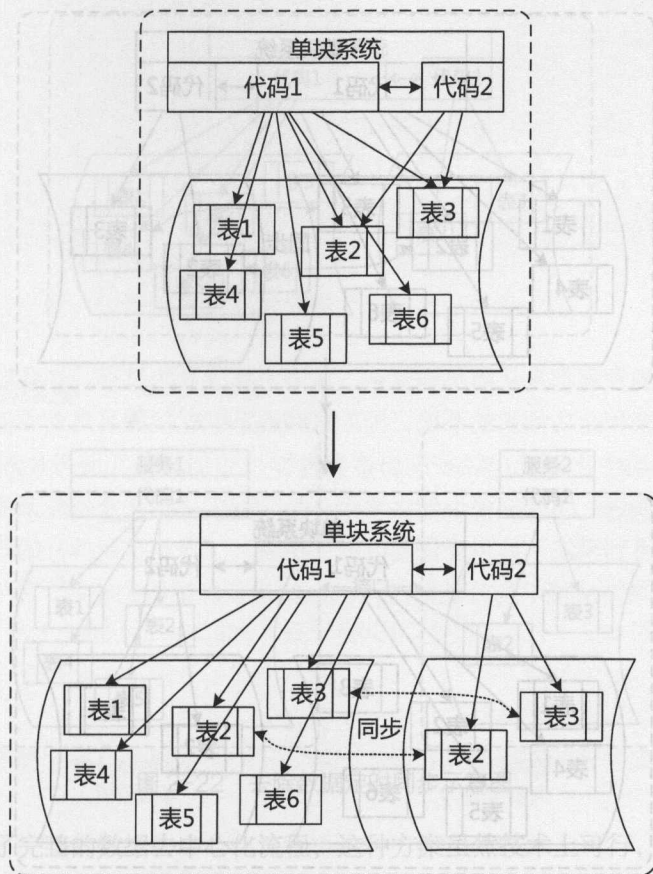


图 2-20 重复数据库模式示意图

### (3) 迁移数据读写操作

完成数据冗余之后，我们将针对数据库写入和读取操作做单独的抽离（见图 2-21）。一般而言，数据写入操作比较容易抽离，因为针对某个业务数据写入的源头通常只有一个，只要明确这个数据源即可。但数据读取就没有那么简单，在没有进行专门的数据拆分之前，同一份数据可能通过各种方式被很多业务所共享，会出现大量跨表查询、跨库查询和技术耦合场景，这时候就需要采用前文所述的各种解决方案进行读操作的分离。这是整个数据去中心化过程中最复杂、难度最大的一步。

在明确服务的边界，业界关于服务边界的划分通常采用面向领域的设计方法，通过识别业务领域、梳理各个子域之间的关系以及明确界限上下文，我们可以充分利用领域驱动设计中的命名概念与方法，并结合服务边界划分的各项原则完成服务边界的划分。

本章最后还讨论了数据与数据之间的关系，针对规范化数据模型存在的问题，提出数据去中心化设计思路，并针对典型场景给出数据去中心化的具体解决方案和工作开展流程。



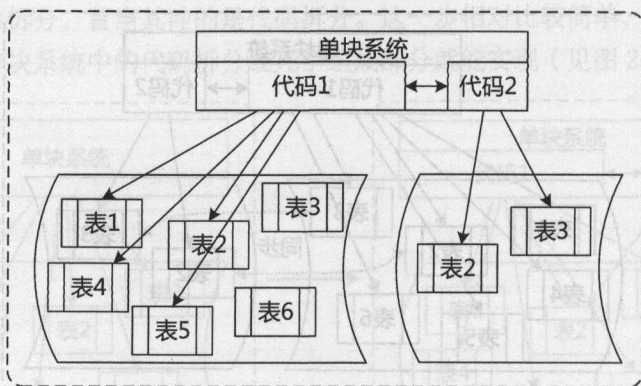
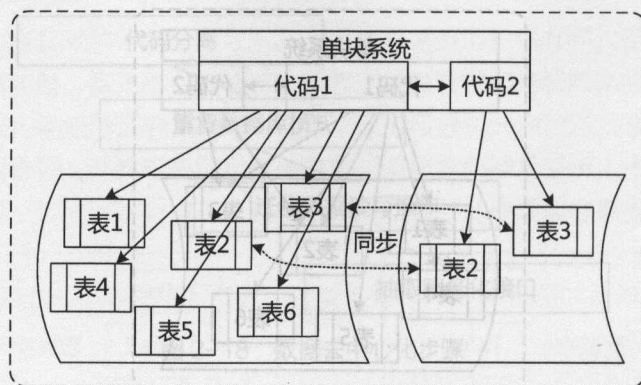


图 2-21 迁移数据读写操作示意图

#### (4) 抽取服务化接口

在完成数据读写操作的迁移、去除数据定时同步机制之后，我们就可以对单块系统中的代码进行拆分，从而实现服务化。但在微服务架构中，我们的目标在于将所有的数据访问统一通过服务化的接口进行，所以抽取服务化接口是完成数据去中心化的最后一个环节（见图 2-22）。

在微服务架构中，数据去中心化是一个重要的目标。为了实现这一目标，我们需要将数据访问统一通过服务化的接口进行。这通常涉及到抽取服务化接口，将数据访问逻辑封装在接口中，从而实现数据去中心化。图 2-20 所示。



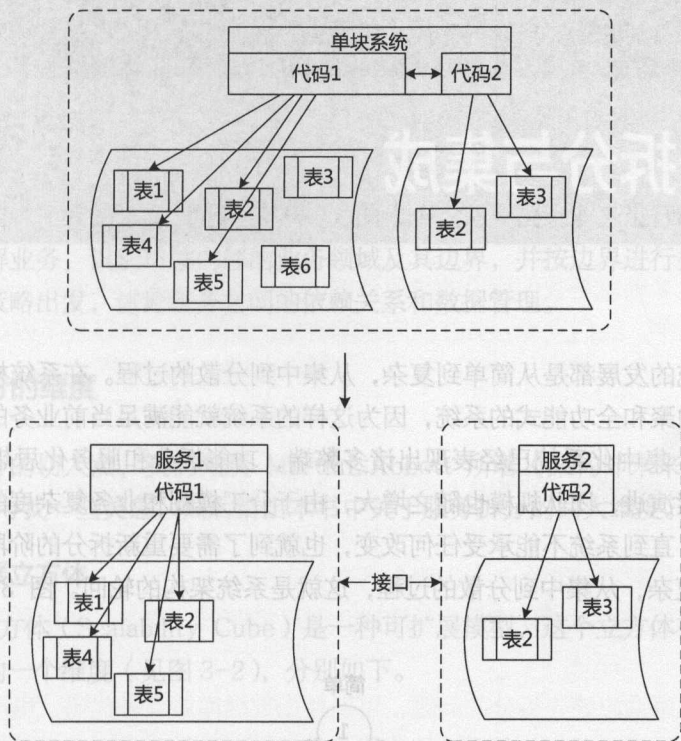


图 2-22 去除数据定时同步示意图

以上步骤构成了完整的数据去中心化流程，这种方案虽然技术上可行，但由于涉及复杂的数据迁移工作，人力和时间成本都很大。这也是数据库去中心化的难度所在。

## 2.5 本章小结

本章介绍了微服务的建模方式，服务建模方式的确定是后续进行服务拆分、集成以及围绕服务开展团队协作的基础。

服务建模首先需要明确服务的分类和模型。服务分类侧重于梳理服务与业务之间的关系并抽象出不同类别和层次的服务体系，而服务模型则用于表述服务。

服务建模最具挑战性的工作是明确服务的边界，业界关于服务边界的划分通常采用面向领域的设计方法。通过识别业务领域、梳理各个子域之间的关系以及明确界限上下文，我们可以充分利用领域驱动设计中的各个概念与方法，并结合服务边界划分的各项原则完成服务边界的划分。

本章最后还讨论了服务与数据之间的关系，针对规范化数据模型存在的问题，提出数据去中心化设计思想，并结合典型场景给出数据去中心化的具体解决方案和工作开展流程。



## 第3章

# 服务拆分与集成

任何软件系统的发展都是从简单到复杂，从集中到分散的过程。在系统构建初期，我们习惯于构建单一、内聚和全功能式的系统，因为这样的系统就能满足当前业务的需求。而当系统发展到一定阶段，集中化系统已经表现出诸多弊端，功能拆分和服务化思想和实践就会被引入。而当系统继续演进，团队规模也随之增大，由于分工模糊和业务复杂度的不断上升，系统架构逐渐被腐化，直到系统不能承受任何改变，也就到了需要重新拆分的阶段。推倒重来意味着重复从简单到复杂，从集中到分散的过程，这就是系统架构的轮回。图 3-1 展示了这一轮回的表现形式。

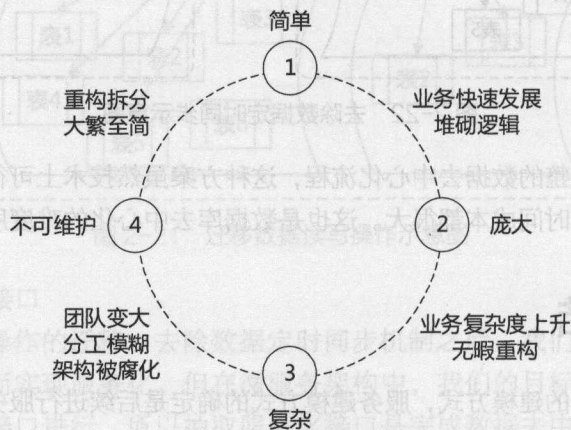


图 3-1 架构的轮回

架构轮回给我们的一大启示就在于将所有东西放在一个系统中是不好的，软件系统的关注点应该清晰划分，并能通过功能拆分降低系统复杂性。系统架构的轮回不可避免，我们能做的就是尽量降低架构被腐化的程度和速度，而微服务架构的提出恰恰就是为了实现这一目标。

在服务拆分之前，我们需要对服务本身进行建模，服务建模能够明确服务的边界以及集成策略，这部分内容已经在上一章中得到阐述。本章将在服务建模的基础上，具体分析服务拆分



的策略和手段，同时也给出对拆分之后的服务进行集成的各种实现方法和技术体系。

## 3.1 服务拆分

在微服务架构中，我们认为服务是业务能力的代表，需要围绕业务进行组织。服务拆分的关键在于正确理解业务，识别单体内部的业务领域及其边界，并按边界进行拆分。本节将从服务拆分的维度和策略出发，讨论服务之间的依赖关系和数据管理。

### 3.1.1 服务拆分的维度

关于服务拆分的切入点，我们先从 Martin L. Abbott 所著《架构即未来》中介绍的 AKF 扩展立方体<sup>①</sup>出发寻找一些灵感，然后给出本书中关于服务拆分的两大维度。

#### 1. AKF 扩展立方体

AKF 扩展立方体（Scalability Cube）是一种可扩展模型，这个立方体有三个轴线，每个轴线描述扩展性的一个维度（见图 3-2），分别如下。

- X 轴  
代表无差别的克隆服务和数据，工作可以很均匀地分散在不同的服务实例上。
- Y 轴  
关注应用中职责的划分，如数据类型、交易执行类型的划分。
- Z 轴  
关注服务和数据的优先级划分，如分地域划分。

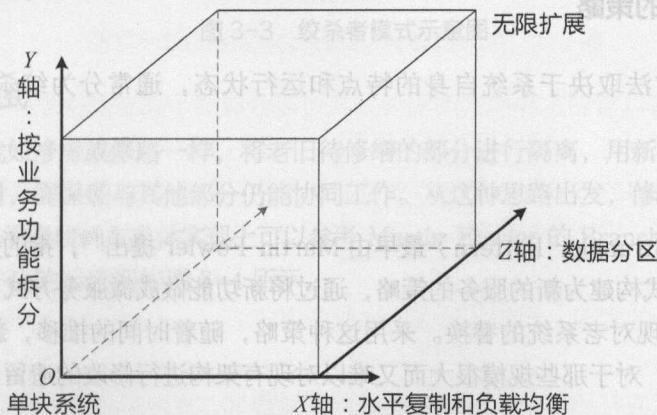


图 3-2 AKF 扩展立方体



以上  $X$ 、 $Y$  和  $Z$  轴的划分可以概括为  $X$  轴关注水平复制， $Z$  轴类似数据分区，而  $Y$  轴则强调基于不同的业务进行拆分。理论上按照这三个扩展维度，可以将一个单体系统进行无限扩展。例如，用户预约挂号应用，一个集群撑不住时，分了多个集群，后来用户激增还是不够用，经过分析发现是用户和医生访问量很大，那么就将预约挂号应用拆成了患者服务、医生服务、支付服务等三个服务。三个服务的业务特点各不相同，独立维护，各自都可以再次按需扩展。

在图 3-2 中， $Y$  轴就是我们所说的微服务的拆分模式，即基于不同的业务进行拆分。但在进行业务拆分过程中，我们发现业务往往与数据有较大耦合性，所以接下去我们把业务和数据结合起来对服务拆分的维度展开讨论。

## 2. 业务与数据

服务拆分存在两大维度，即业务与数据。业务体现在各种功能代码中，通过确定业务的边界，并使用领域与界限上下文、领域事件等技术手段可以实现拆分。而数据的拆分则体现在如何将集中式的中心化数据转变为各个微服务各自拥有的独立数据，这部分工作同样十分具有挑战性。关于业务拆分和数据拆分的方法及实践我们已经在上一章 2.3 节中做了全面介绍，这里不再赘述。

关于业务和数据谁应该先拆分的问题，可以是先数据库后业务代码，也可以是先业务代码后数据库。然而在拆分中遇到的最大挑战可能会是数据层的拆分，因为在数据库中，可能会存在各种跨表连接查询、跨库连接查询以及不同业务模块的代码与数据耦合得非常紧密的场景，这会导致服务的拆分非常困难。因此在拆分步骤上我们更多地推荐数据库先行。数据模型能否彻底分开，很大程度上决定了微服务的边界功能是否彻底划清。

### 3.1.2 服务拆分的策略

服务拆分的方法取决于系统自身的特点和运行状态，通常分为绞杀者与修缮者两种模式。

#### 1. 绞杀者模式

绞杀者模式 (Strangler Pattern) 最早由 Martin Fowler 提出<sup>[10]</sup>，指的是在现有系统外围将新功能用新的方式构建为新的服务的策略，通过将新功能做成微服务方式，而不是直接修改原有系统，逐步实现对老系统的替换。采用这种策略，随着时间的推移，新的服务就会逐渐“绞杀”老的系统。对于那些规模很大而又难以对现有架构进行修改的遗留系统，推荐采用绞杀者模式。

绞杀者模式的示意图如图 3-3 所示，我们可以看到随着功能演进和时间的不断推移，老



的遗留系统功能被逐步削弱，而采用微服务架构的新功能越积越多，最终会形成从量变到质变的过程。绞杀者模式在具体实施过程中，所需要把握的最主要原则就是对于任何需要开发的功能一定要完整地采用微服务架构。对于完全独立的新功能，这一点比较容易把握，而对于涉及老业务变更的新功能，则需要通过重构达到这一目标。

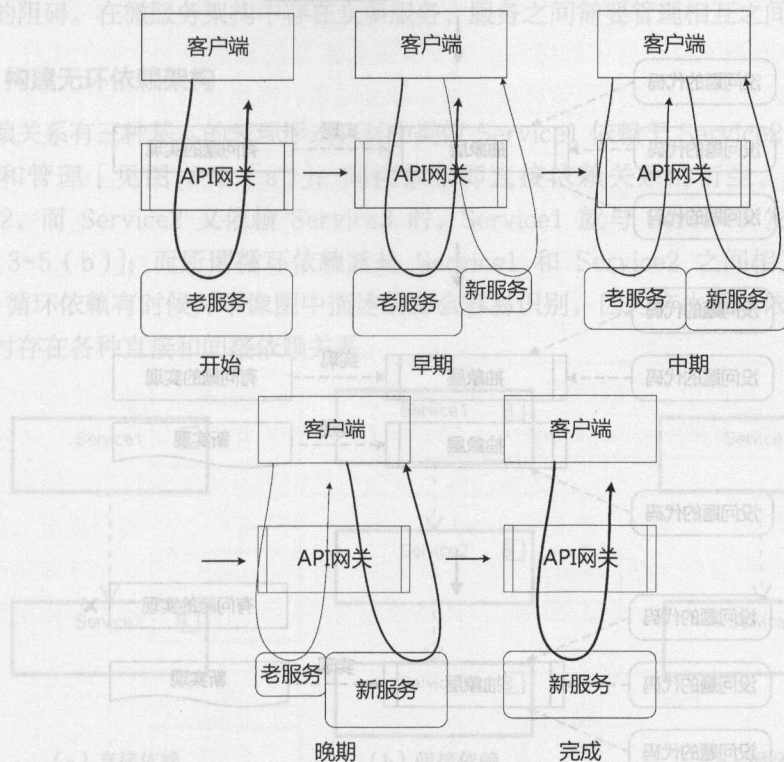


图 3-3 绞杀者模式示意图

## 2. 修缮者模式

修缮者模式就如修房或修路一样，将老旧待修缮的部分进行隔离，用新的方式对其进行单独修复。修复的同时，需保证与其他部分仍能协同工作。从这种思路出发，修缮者模式更多表现为一种重构技术。修缮者模式在具体实现上可以参考 Martin Fowler 的 BranchByAbstraction 重构方法<sup>[11]</sup>，该重构方法的示意图如图 3-4 所示。

Order 之间存在依赖关系。左方各模块采用旧实现，右方的模块则是新实现。

根据无环依赖原则（No Cycle Dependencies）

然而从，考虑到完全剥离旧实现并重新实现，对系统运行，其期望是那个各的存取权限实现的实现来

环依赖，该条原则对于微服务架构而言同样

图 3-6 循环依赖示例



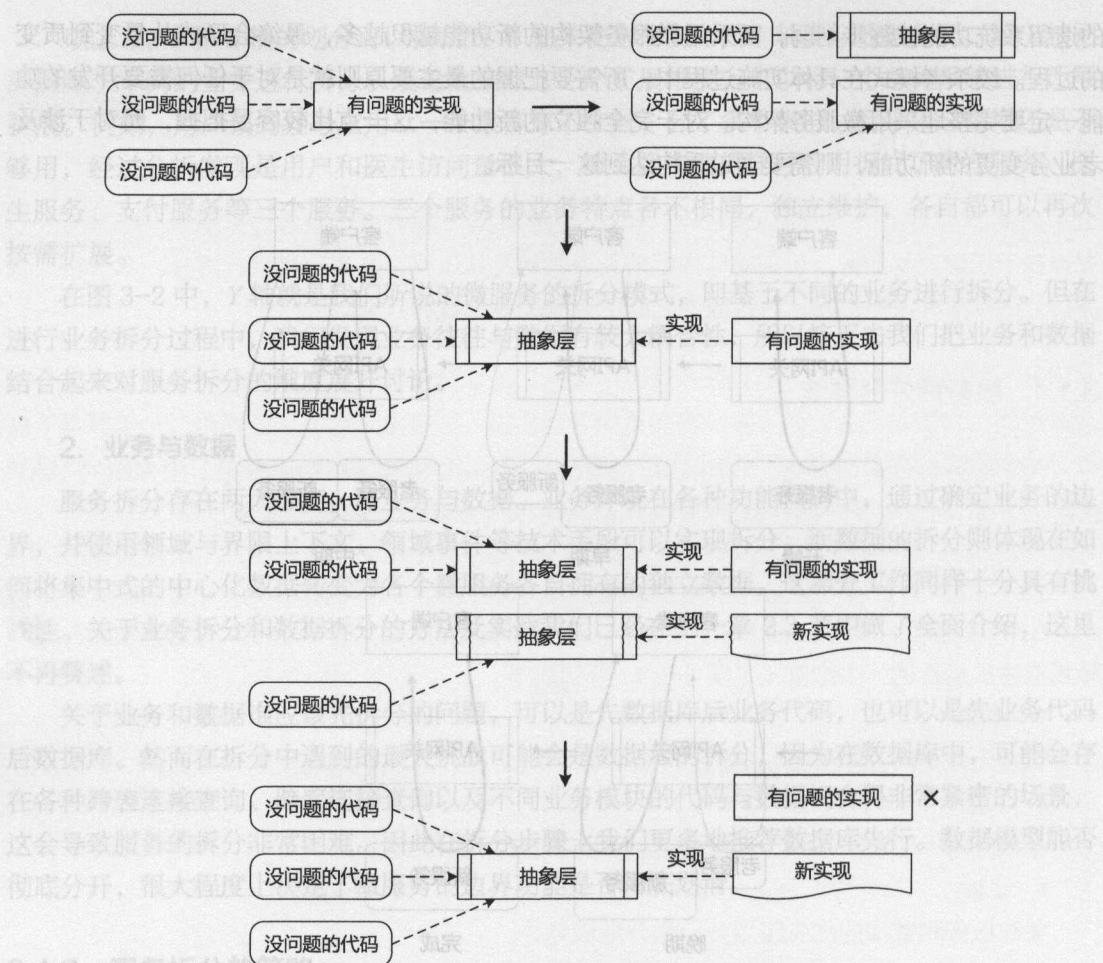


图 3-4 BranchByAbstraction 重构方法示意图

从图 3-4 中，可以看到这种模式的实现方式可以分成三个主要步骤。

#### • 抽象层提取

首先通过识别内部的待拆分功能，对其增加抽象接口层，同时对原有代码进行改造，确保其同样实现该抽象层。这样在依赖关系上就添加了一个中间层。

#### • 抽象层实现

为抽象层提供新的实现，新的实现采用微服务方式。

#### • 抽象层替换

采用新的实现对原有的各个抽象层实现进行逐步替换，直至原有实现被完全废弃，从而完成新老实现方式之间的替换。



### 3.1.3 管理服务的依赖关系

在系统的各种组件之间,尤其是类、包、模块以及服务之间都可能存在依赖关系。依赖在某种程度上不可避免,但是过多地依赖势必会增加系统复杂性和降低代码维护性,从而成为团队开发的阻碍。在微服务架构中存在众多服务,服务之间需要管理相互之间的依赖关系。

#### 1. 构建无环依赖架构

依赖关系有三种基本的表现形式,其中类似 Service1 依赖于 Service2 这种直接依赖最容易识别和管理[见图 3-5(a)];间接依赖即直接依赖关系的衍生,当 Service1 依赖 Service2,而 Service2 又依赖 Service3 时,Service1 就与 Service3 发生了间接依赖关系[见图 3-5(b)];而所谓循环依赖就是 Service1 和 Service2 之间相互依赖[见图 3-5(c)],循环依赖有时候并不像图中描述的那么容易识别,因为产生循环依赖的多个组件之间可能同时存在各种直接和间接依赖关系。

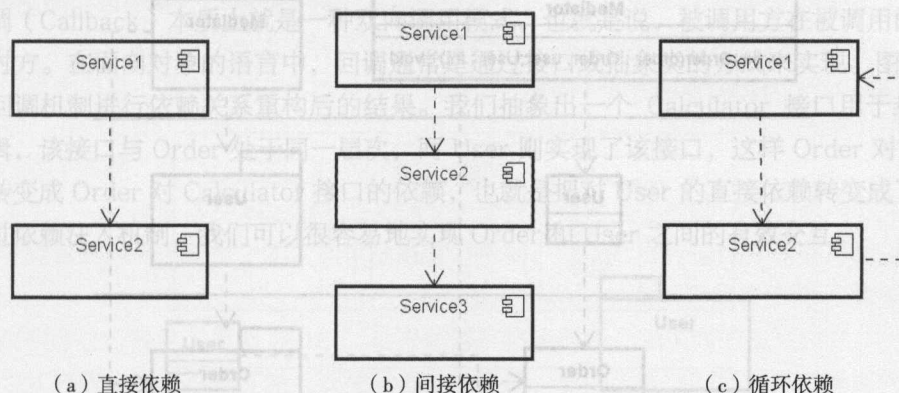


图 3-5 服务依赖的三种基本关系

图 3-6 就是一个循环依赖的例子, User 对象可以创建 Order 对象并保持 Order 对象列表,而 Order 对象同样需要使用 User 对象,并根据 User 对象中的打折(Discount)信息计算 Order 金额,这样对象 User 和 Order 之间就存在循环依赖关系。

根据无环依赖原则(Acyclic Dependencies Principle, ADP),系统设计中不应该存在循环依赖,该条原则对于微服务架构而言同样

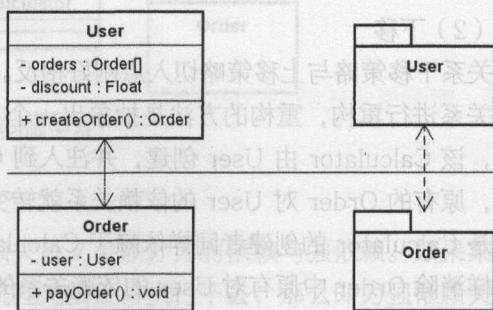


图 3-6 循环依赖示例



适用。消除循环依赖的基本思路就是通过在两个相互循环依赖的组件之间添加中间层, 变循环依赖为间接依赖。有三种策略可以做到这一点, 分别是上移、下移和回调<sup>[12]</sup>。下面我们将通过基于组件的具体示例介绍这三种策略, 因为服务由一个或多个组件构成, 所以其适用于服务之间的关系管理。

### (1) 上移

关系上移意味着把两个相互依赖组件中的交互部分抽象出来形成一个新的组件, 而新组件同时包含着原有两个组件的引用, 这样就把循环依赖关系剥离出来并上升到一个更高层次的组件中。图 3-7 就是使用上移策略对 User 和 Order 原始关系进行重构的结果, 我们引入了一个新的组件 Mediator, 并通过其提供的 payOrder 方法对循环依赖进行了剥离, 该方法同时使用 Order 和 User 作为参数并实现了 Order 中根据 User 的打折信息进行金额计算的逻辑。Mediator 组件消除了 Order 中原有对 User 的依赖关系并在依赖关系上处于 User 和 Order 的上层。

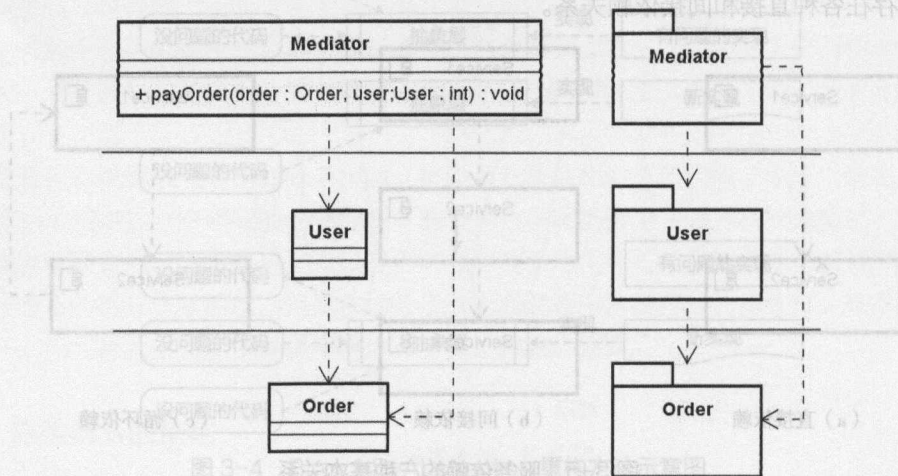


图 3-7 上移关系

### (2) 下移

关系下移策略与上移策略切入点刚好相反。我们同样针对图 3-6 中 User 和 Order 的循环依赖关系进行重构, 重构的方法是抽象出一个 Calculator 组件专门包含打折信息的金额计算方法, 该 Calculator 由 User 创建, 并注入到 Order 的 pay 方法中去 (见图 3-8)。通过这种方式, 原有的 Order 对 User 的依赖关系就转变为 Order 对 Calculator 的依赖关系, 而 User 因为是 Calculator 的创建者同样依赖于 Calculator, 这种生成一个位于 User 和 Order 之下但能同样消除 Order 中原有对 User 的依赖关系的组件的策略, 就称之为下移。



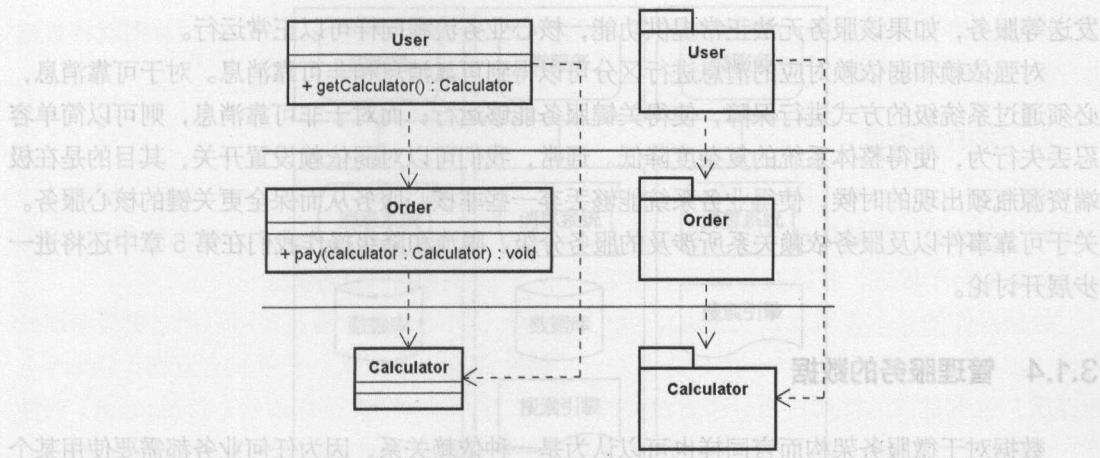


图 3-8 下移关系

### (3) 回调

回调（Callback）本质上就是一种双向调用模式，也就是说，被调用方在被调用的同时也会调用对方。在面向对象的语言中，回调通常是通过接口或抽象类的方式来实现。图 3-9 就是通过回调机制进行依赖关系重构后的结果。我们抽象出一个 Calculator 接口用于封装金额计算逻辑，该接口与 Order 处于同一层次，而 User 则实现了该接口，这样 Order 对 User 的依赖就转变成 Order 对 Calculator 接口的依赖，也就是把对 User 的直接依赖转变成了间接依赖。通过依赖注入机制，我们可以很容易地实现 Order 和 User 之间的有效交互。

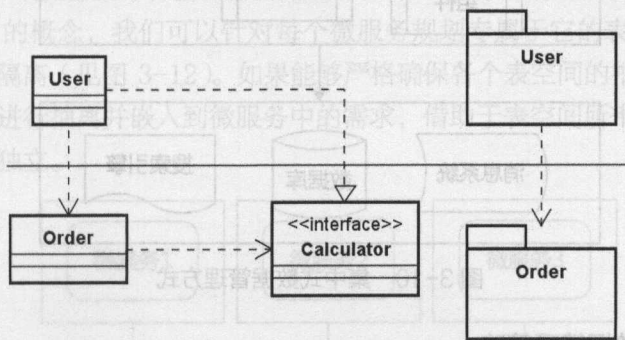


图 3-9 回调关系

## 2. 弱依赖与强依赖

从故障容忍度的角度来看，服务之间的依赖关系又可以分为弱依赖与强依赖。如果某个微服务无法正常提供服务，则依赖它的整个业务流程无法正常执行下去，那么即为强依赖关系，也就是说，强依赖关系是服务正常运转的基本单元。而弱依赖则没有这样的限制，对于如消息



发送等服务, 如果该服务无法正常提供功能, 核心业务流程同样可以正常运行。

对强依赖和弱依赖对应的消息进行区分可以得到可靠消息和非可靠消息。对于可靠消息, 必须通过系统级的方式进行保障, 使得关键服务能够运行; 而对于非可靠消息, 则可以简单容忍丢失行为, 使得整体系统的复杂度降低。通常, 我们可以对弱依赖设置开关, 其目的是在极端资源瓶颈出现的时候, 使得业务系统能够丢弃一些非核心服务从而保全更关键的核心服务。关于可靠事件以及服务依赖关系所涉及的服务分级、限流和降级操作我们在第 5 章中还将进一步展开讨论。

### 3.1.4 管理服务的数据

数据对于微服务架构而言同样也可以认为是一种依赖关系, 因为任何业务都需要使用某个数据容器作为持久化的机制或者数据处理的媒介, 这里的数据容器不仅仅是指关系型数据库, 而是泛指包括消息队列、搜索引擎索引以及各种 NoSQL 在内的数据媒介。微服务架构中存在一种说法, 即我们需要将所有微服务所用的资源全部嵌入到该服务中, 从而确保微服务的独立性。但从数据管理角度出发, 想要完整实现这种资源嵌入并不容易, 因为长久以来, 我们已经习惯了使用集中式的数据管理方式(见图 3-10)。

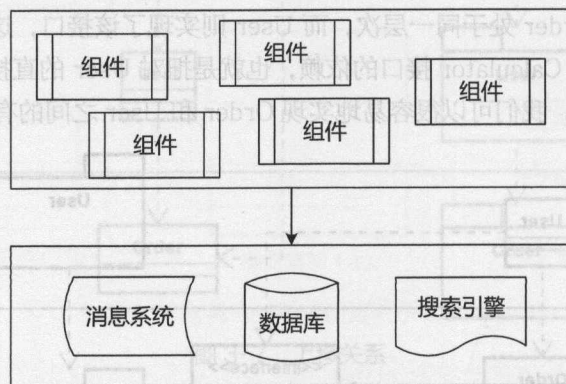


图 3-10 集中式数据管理方式

#### 1. 微服务中的数据管理策略

集中式数据管理方式有其明显的优点, 我们可以通过专业的 DBA 或系统管理员来规划、实施以及优化以关系型数据库为代表的数据存储和处理媒介, 把这部分工作从普通的开发人员中解放出来。但是微服务架构崇尚把数据也嵌入到微服务内部, 这样涉及数据相关的关系型数据库、各种 NoSQL 存储、搜索引擎索引等将成为微服务自身的一部分(见图 3-11)。



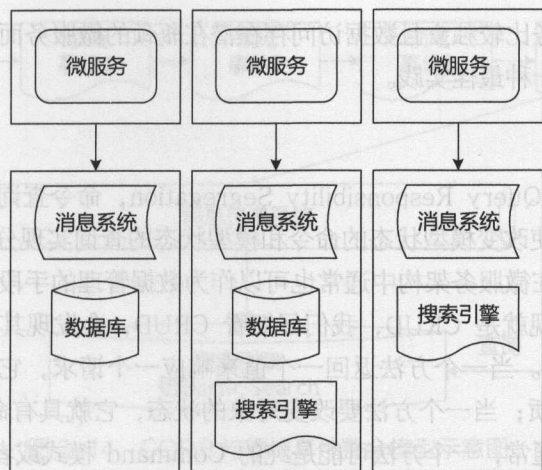


图 3-11 微服务嵌入式数据管理方式

如图 3-11 所示的将数据完全嵌入到微服务中的方式看上去比较完美，但实际上也存在一些问题。一方面，在微服务架构中，我们对各个服务的设计目标是尽量简单和独立，所以对于大多数微服务而言，为每个服务配备独立的数据存储容器显然是一种浪费，因为多数服务都会比较简单；另一方面，很难想象对于一个由很多个微服务构成的系统而言，搜索引擎等工具同样存在很多个实例，这会给运维工作带来巨大挑战。

事实上，一个微服务并不一定需要去独立包含自身所需要的数据，我们也可以采用集中式的数据管理方式在一定程度上实现数据独立性。例如，对于数据存储媒介而言可以设置命名空间（Namespace）的概念，我们可以针对每个微服务规划专属于它的表空间，从而在逻辑上与其他微服务进行隔离（见图 3-12）。如果能够严格确保各个表空间的相互独立，一旦有将数据从集中式容器中进行抽离并嵌入到微服务中的需求，借助于表空间所带来的隔离性，我们也很容易实现数据的独立。

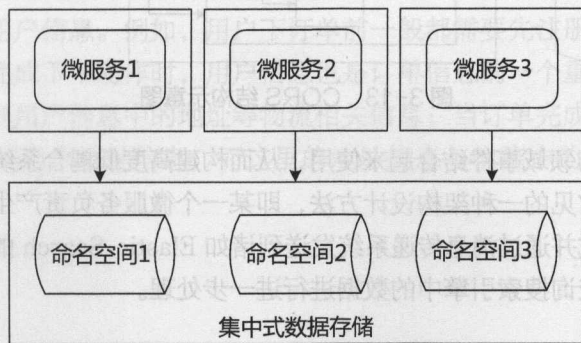


图 3-12 命名空间管理方式



当然，对于一些业务比较独立且数据访问存在潜在瓶颈的微服务而言，在设计时就将数据进行物理隔离自然也是一种最佳实践。

## 2. CQRS 模式

CQRS (Command Query Responsibility Segregation, 命令查询的责任分离) 模式是一种架构体系模式，能够使改变模型状态的命令和模型状态的查询实现分离。CQRS 属于 DDD 应用领域的模式，在微服务架构中通常也可以作为数据管理的手段。

数据操作的基本表现就是 CRUD，我们仔细看 CRUD，会发现其实可以把它更简单地分为读 (R) 和写 (CUD)。当一个方法返回一个值来响应一个请求，它就具有查询 (Query) 的性质，也就是读的性质；当一个方法要改变对象的状态，它就具有命令 (Command) 的性质，也就是写的性质。通常，一个方法可能是纯的 Command 模式或者是纯的 Query 模式，亦或是两者的混合体。在设计接口时，如果可能，应该尽量使接口单一化，严格保证方法的行为是命令或者是查询，这样查询方法不会改变对象的状态，没有副作用，而会改变对象状态的方法不可能有返回值。查询功能和命令功能的分离，有助于系统性能，也有利于系统的安全性。图 3-13 展示了 CQRS 模式的一种表现形式。

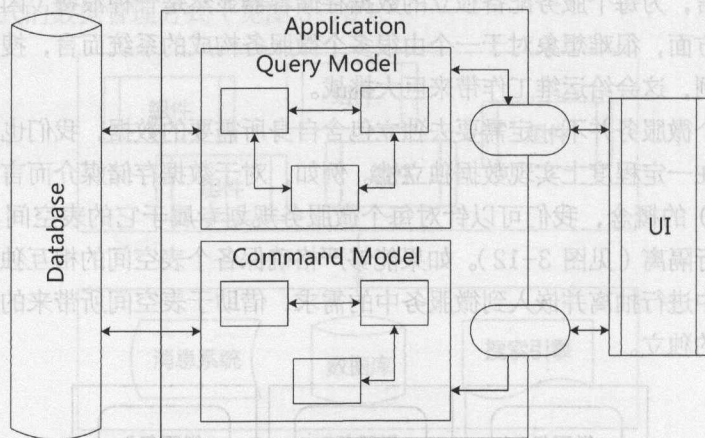


图 3-13 CQRS 结构示意图

CQRS 模式可以与领域事件结合起来使用，从而构建高度低耦合系统。图 3-14 展示的是在互联网系统中非常常见的一种架构设计方法，即某一个微服务负责产生和维护数据。这些数据以事件作为表现形式并通过消息传递系统发送到诸如 Elastic Search 的搜索引擎中，而另一个微服务则专门负责查询搜索引擎中的数据进行进一步处理。



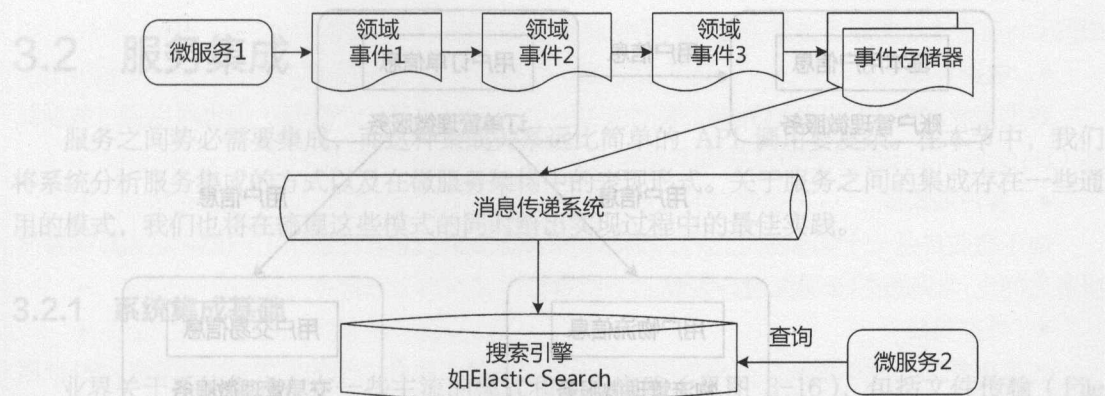


图 3-14 CQRS 与领域事件整合模型示意图

使用 CQRS 模式实际上与数据去中心化是一种互补的关系。正如图 3-13 和图 3-14 所示的 CQRS 基本结构和应用场景，我们会发现把各个服务所依赖的数据统一存储到专门用于查询的中央仓库中也是一种最佳实践。

### 3.1.5 管理事务的边界

在 2.3 节中，我们讨论了服务的边界问题，并提出了界限上下文和聚合等概念。界限上下文关注于各个上下文之间的交互，而聚合则是一个上下文中一组实体或值对象的组合。

在聚合中，我们需要确保一个事务中只修改一个聚合实例，也就是说只能在一个聚合结构上执行原子操作，但不支持跨越多个聚合的 ACID 事务。尽管实际应用中大多数原子操作都可以局限于聚合结构内部，从而确保聚合内部的各个对象之间具有强一致性。但当需要跨越多个聚合时，该如何处理事务呢？

图 3-15 给出了一个现实中的场景，当处理用户订单时，围绕订单业务所展开的各个环节中都可能需要用到用户信息。例如，用户下订单前一般都需要先注册，注册中用到了用户的基本信息；当用户完成下单动作时，用户信息也是订单信息的一个重要组成部分；当订单发货时，肯定需要用到用户信息中的地址等物流相关信息；当订单完成后，需要向用户发送发票时，同样需要用到用户地址信息，而这里的地址信息与物流中的地址信息可能是不一样的。

#### • 接口集成

接口集成是服务之间集成的最常见手段，通常基于业务逻辑的需要进行集成。RPC、



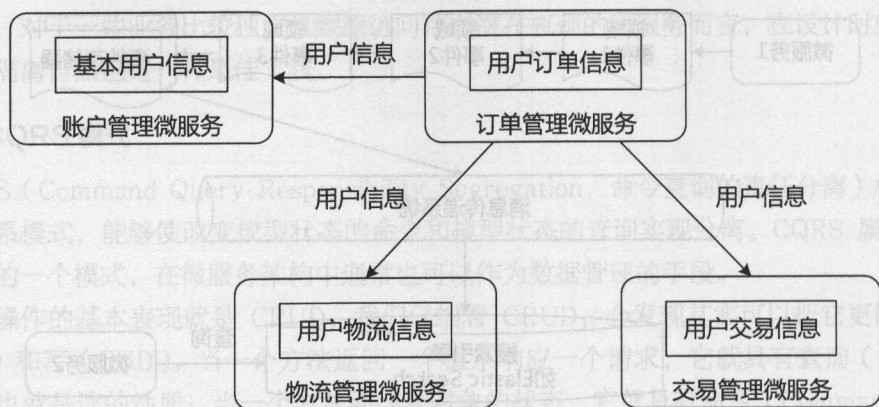


图 3-15 用户数据与多个服务

根据图 3-15 中的业务结构，我们一般会拆分成若干个微服务来处理整个业务流程，如包含用户注册功能的用户账户管理服务、用户订单服务、用户物流服务等，这些服务都用到了用户信息。针对这些场景，我们首先关注的是服务的隔离性，通过引入防腐层（Anti-corruption Layer）<sup>[13]</sup>等架构模式对服务与服务之间相互影响的程度进行管理。另外，我们也要引出一个重要概念，即数据一致性。

数据一致性（Data Consistency）可以分成两种，即强一致性（Strong Consistency）和弱一致性（Weak Consistency）。所谓弱一致性，就是在某个时刻数据可能非一致，但是到达某个时间点以后总能保持一致，也即我们所说的最终一致性（Eventually Consistency）。而强一致性就是需要保证数据的一致性实时的，每一时刻都保持一致。在图 3-15 中，强一致性表现在用户账户管理服务、用户订单服务、用户物流服务等各个服务中的用户信息在任何时候都是一样的，而弱一致性则认为这些服务之间的数据可能存在一定时间的不一致，我们只要确保最终数据能达到一致状态即可。当然，这里所说的“一定时间”肯定也是一个秒级的时间维度。

从事务的角度讲，我们希望用户信息在用户账户管理服务、用户订单服务、用户物流服务等各个服务中都能进行事务管理，实现分布式环境下的数据同步。业界关于分布式事务的实现方式包括经典的两阶段提交和三阶段提交等方案，这些方案能解决一部分数据一致性的问题，但在微服务架构中，传统分布式事务并不是实现数据一致性的最佳选择。在微服务架构中，我们推崇的是打破事务的边界，实现数据的弱一致性。

关于分布式事务存在的问题以及数据一致性相关更详细的探讨可参考本书 5.2 节中的相关内容。



## 3.2 服务集成

服务之间势必需要集成，而这种集成关系远比简单的 API 调用要复杂。在本节中，我们将系统分析服务集成的方式以及在微服务架构中的表现形式。关于服务之间的集成存在一些通用的模式，我们也将梳理这些模式的同时给出实现过程中的最佳实践。

### 3.2.1 系统集成基础

业界关于系统集成存在一些主流的模式和工程实践（见图 3-16），包括文件传输（File Transfer）、共享数据库（Shared Database）、远程过程调用（RPC）和消息传递（Messaging）。

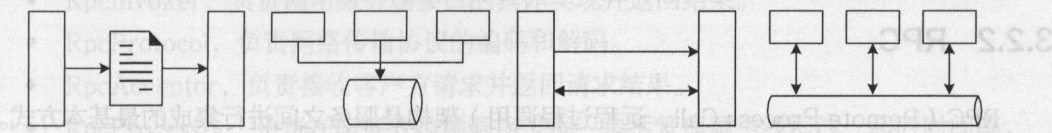


图 3-16 系统集成基本模式

以上四种主流的集成模式各有优缺点。文件传输方式最大的挑战在于如何进行文件的更新和同步；如果使用数据库，在多方共享的条件下如何确保数据库模式统一是一个大问题；RPC 容易产生瓶颈节点；而消息传递在提供松耦合的同时也加大了系统的复杂性。RPC 和消息传递面对的都是分布式环境下的远程调用，远程调用区别于内部方法调用，一方面网络不一定可靠和存在延迟问题，另一方面集成通常面对的是一些异构系统。

对于微服务架构而言，我们的思路是尽量采用标准化的数据结构并降低系统集成的耦合度。我们会根据需要采用如图 3-16 所示的四种典型的系统集成模式，同时还会引入一些其他手段来达到服务与服务之间的有效集成。我们把微服务架构中服务之间的集成模式分为以下四大类（见图 3-17）。

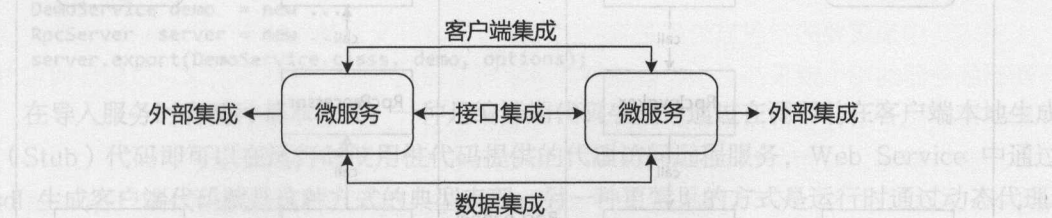


图 3-17 微服务集成的四种主要方式

- 接口集成

接口集成是服务之间集成的最常见手段，通常基于业务逻辑的需要进行集成。RPC、



REST、消息传递和服务总线都可以归为这种集成方式。

- 数据集成

数据集成同样可以用于微服务之间的交互，共享数据库是一个选择，但也可以通过数据复制（Data Replication）的方式实现数据集成。

- 客户端集成

由于微服务是一个能够独立运行的整体，有些微服务会包含一些 UI 界面，这也意味着微服务之间也可以通过 UI 界面进行集成。

- 外部集成

这里把外部集成单独剥离出来的原因在于现实中很多服务之间的集成需求来自于与外部服务的依赖和整合，而在集成方式上也可以综合采用接口集成、数据集成和 UI 集成。

### 3.2.2 RPC

RPC（Remote Process Call，远程过程调用）架构是服务之间进行集成的最基本方式。

#### 1. RPC 结构

我们可以对 RPC 架构进行剖析，得到图 3-18 的结构图，该结构图包括了微服务之间在分布式环境下交互时所需的各个基本功能组件。

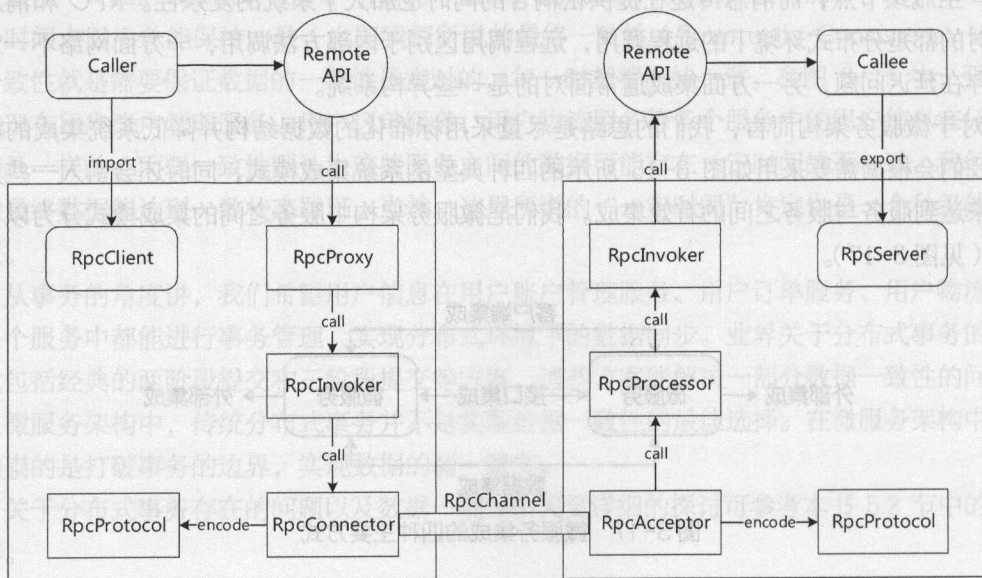


图 3-18 RPC 结构



从图 3-18 中，可以看到 RPC 架构由左右对称的两大部分构成，分别代表了一个远程过程调用的客户端和服务端组件。客户端组件与职责如下。

- RpcClient，负责导入（Import）由 RpcProxy 提供的远程接口的代理实现。
- RpcProxy，远程接口的代理实现，提供远程服务本地化访问的入口。
- RpcInvoker，负责编码和发送调用请求到服务方并等待结果。
- RpcProtocol，负责网络传输协议的编码和解码。
- RpcConnector，负责维持客户端和服务端连接通道和发送数据到服务端。
- RpcChannel，网络数据传输通道。

而服务端组件与职责如下。

- RpcServer，负责导出（Export）远程接口。
- RpcInvoker，负责调用服务端接口的具体实现并返回结果。
- RpcProtocol，负责网络传输协议的编码和解码。
- RpcAcceptor，负责接收客户方请求并返回请求结果。
- RpcProcessor，负责在服务方控制调用过程，包括管理调用线程池、超时时间等。
- RpcChannel，网络数据传输通道。

## 2. RPC 实现

在 RPC 架构实现思路，远程服务提供者以某种形式提供服务调用相关信息，远程代理对象通过动态代理拦截机制生成远程服务的本地代理，让远程调用在使用上就如同本地调用一样。而网络通信应该与具体协议无关，通过序列化和反序列化方式对网络传输数据进行有效传输。

在基于分布式环境的交互过程中，远程服务的导入和导出需要遵循特定的业务接口，确保双方在通信语义上的一致。假设我们使用 DemoService 作为统一业务接口，当进行服务导出时，可以使用如下的代码风格：

```
DemoService demo = new ...;  
RpcServer server = new ...;  
server.export(DemoService.class, demo, options);
```

在导入服务时有两种基本方式，一种是编译期代码生成，通过在调用前在客户端本地生成桩（Stub）代码即可以在运行时使用桩代码提供的代理访问远程服务，Web Service 中通过 wsdl 生成客户端代码就是这种方式的典型表现；另一种更常见的方式是运行时通过动态代理/字节码的方式动态生成代码。对 DemoService 服务进行导入的表现形式如下：

```
RpcClient client = new ...;  
DemoService demo = client.refer(DemoService.class);  
demo.hi("how are you?");
```



RPC 架构在实现上有很多具体的工具和技术体系。例如，基于 Web Service 的 SOAP 和 XML-RPC，以及 Google gRPC、Facebook Thrift 和 Alibaba Dubbo 等框架。

### 3.2.3 REST

REST (Representational State Transfer, 表述性状态转移) 从技术上讲也可以认为是 RPC 架构的一种具体表现形式，因为 RPC 架构中最基本的网络通信、序列化/反序列化、传输协议和服务调用等组件都能在 REST 中有所体现。但 REST 代表的并不是一种技术，也不是一种标准和规范，而是一种设计风格。基于这个风格设计的软件可以更简洁，更有层次，更易于实现缓存等机制。

要理解 RESTful 架构，最好的方法就是去理解它的全称 Representational State Transfer 这个词组，直译过来就是“表现层状态转移”，其实它省略了主语。“表现层”其实指的是“资源”的“表现层”，所以 REST 通俗来讲就是：资源在网络中以某种表现形式进行状态转移。分解开来，可以得到三个更加明确的概念。

- Resource

资源，即数据，如 User、Order 等。

- Representational

某种表现形式，如 JSON、XML、JPEG。

- State Transfer

状态变化，通过 HTTP 动词实现。

#### 1. REST 基本理念

REST 提出了一组架构约束条件和原则。满足这些约束条件和原则的应用程序或设计就是 RESTful。现实世界中的事物都可以被认为是一种资源，我们可以根据这些原则设计以资源为中心的服务。

REST 中最重要的一条原则就是客户端和服务端之间交互的无状态 (Stateless) 性。从客户端到服务器的每个请求都必须包含理解该请求所必需的信息，无状态请求可以由任何可用服务器实现响应，十分适合微服务架构的运行环境。

在服务器端，应用程序状态和功能可以分为各种资源。每个资源都使用 URI (Universal Resource Identifier, 统一资源标识符) 得到一个唯一的地址。所有资源都共享统一的接口，以便在客户端和服务端之间传输状态。在传输协议上使用的就是标准的 HTTP 方法，比如最常见的 GET、PUT、POST 和 DELETE。

REST 在实现上的关键是定义可表示流程元素或资源的对象。在 REST 中，每一个对象



都是通过 URI 来表示，对象负责将状态信息打包进每一条消息内，以保证对象处理的无状态性。表 3-1 所展示的就是针对用户（User）这个对象所能代表的各种资源 URI 及其表述。

表 3-1 RESTful 风格示例

URL	HTTP 方法	描 述
http://www.example.com/users	GET	获取 User 对象列表
http://www.example.com/users	PUT	更新一组 User 对象
http://www.example.com/users	POST	新增一组 User 对象
http://www.example.com/users	DELETE	删除所有 User
http://www.example.com/users/tianyalan	GET	根据用户名 tianyalan 获取 User 对象
http://www.example.com/users/tianyalan	PUT	根据用户名 tianyalan 更新 User 对象
http://www.example.com/users/tianyalan	POST	添加用户名为 tianyalan 的新 User 对象
http://www.example.com/users/tianyalan	DELETE	根据用户名 tianyalan 删除 User 对象

另一个需要注意的点在于 RESTful 风格下执行的 HTTP 请求是一个同步操作，通常一个服务发送请求并等待响应，当在网络传输过程中产生较长延迟时这可能会是一个问题，因为这会导致对该服务产生依赖的其他服务也会处于一直等待状态。当等待时间过长时，我们需要有一个超时机制来终止本次请求。请求长时间不响应的原因可能是服务发生错误或者网络出现问题，通过引入合适的超时机制能够提高服务的可靠性。在微服务架构中，因为服务之间存在直接或间接的依赖，提高服务可靠性的更重要的意义在于当一个服务出现问题时，不会传播到其他服务从而导致整个系统出现服务可用性问题。关于服务可靠性更多的讨论我们将放在第 5 章中进一步展开。

REST 中还有一个成熟度的概念，当谈及 REST 成熟度时，常常会引用 Richardson 所提出来的 REST 成熟度模型（Maturity Model）<sup>[14]</sup>，并视之为正确的度量方法。REST 成熟度模型如图 3-19 所示。

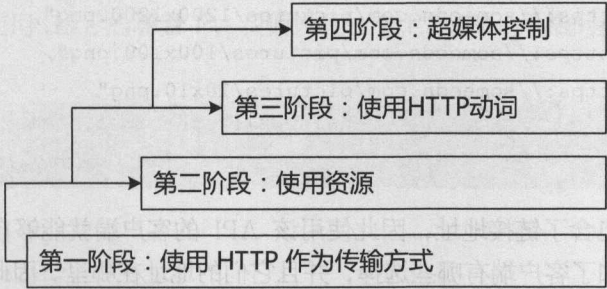


图 3-19 REST 成熟度模型



从图 3-19 中，我们明确看到 REST 的成熟度可以分为四个不同的层次，分别如下。

- Level 0

Web 服务只是使用 HTTP 作为传输方式。

- Level 1

Web 服务引入了资源的概念。

- Level 2

Web 服务使用不同的 HTTP 方法进行不同的操作。

- Level 3

Web 服务使用 HATEOAS。

HATEOAS (Hypermedia as the Engine of Application State, 基于超媒体的应用状态引擎) 是整个模型中的最高层次, 也是 REST 中的一个重要组件, 接下去我们将对其做深入分析。

## 2. HATEOAS

### (1) HATEOAS 基本概念

要解释 HATEOAS 这个概念先要解释什么是超媒体。我们已经知道什么是多媒体 (Multimedia), 以及什么是超文本 (Hypertext)。其中超文本特有的优势是拥有超链接 (Hyperlink)。如果我们把超链接引入到多媒体中, 那就得到了超媒体, 因此关键角色还是超链接。使用超媒体作为应用引擎状态, 意思就是应用引擎的状态变更由客户端访问不同的超媒体资源驱动。

例如, 我们看如下的服务请求响应结果:

```
GET https://api.example.com/profile
{
  "name": "tianyalan",
  "picture": {
    "large": "https://somecdn.com/pictures/1200x1200.png",
    "medium": "https://somecdn.com/pictures/100x100.png",
    "small": "https://somecdn.com/pictures/10x10.png"
  }
}
```

由于在响应中包含了链接地址, 因此使用该 API 的客户端就能够自由选择要下载怎样的图片。这些链接告知了客户端有哪些选择, 并且它们的地址在哪里。因此在这里我们无需同时返回三个不同版本的用户档案图片, 我们所做的只是告诉客户端有三种可用的图片尺寸可以选



择，并且告诉客户端能够在哪里找到这些图片。这样一来，客户端就能够根据不同的场景，做出符合自身需要的选择。而且，如果客户端只需要一种格式的图片，那就无需下载全部三种版本的图片。因此，这种表现形式有很多优势，既减少了网络负载，又增进了客户端的灵活性，更增进了 API 的可探索性。

HATEOAS 的重要性在于打破了客户端和服务端之间严格的契约，使得客户端可以更加智能和自适应，而 REST 服务本身的演化和更新也变得更加容易。我们知道，基于 HTTP 方法的 REST 风格，客户端需要根据服务器提供的相关文档来了解所暴露的资源和对应的操作。而基于 HATEOAS 的 REST 服务可以实现服务端和客户端最大程度上的解耦，客户端可以通过服务器提供的资源来智能地发现可以执行的操作。

(2) HAL

HATEOAS 更多只是一种概念，而 HAL (Hypertext Application Language, 超文本应用语言) 是 HATEOAS 的一种实现方式。与 REST 不同，对于每个资源 HAL 又将其分成状态 (State)、链接 (Links) 和子资源 (Embedded Resource) 三个标准部分，如图 3-20 所示。这里的状态是指资源本身固有的属性，链接定义了与当前资源相关的一组资源的链接的集合，而子资源描述当前资源的内容，提供嵌套资源的定义。

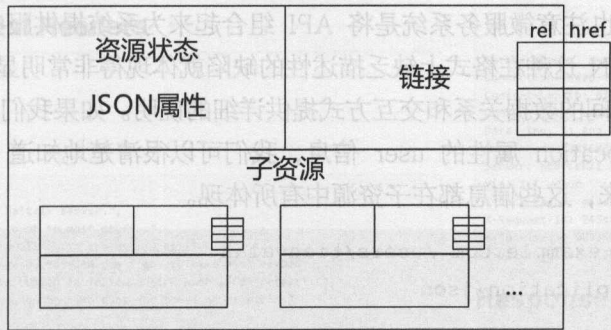


图 3-20 HAL 模型

举例来说，在不使用 HAL 的场景下，我们设计一个 RESTful 风格的接口，一般会采用如下的表现形式：

```
GET http://api.example.com/users/tianyalan
Content-Type: application/json
{
  "id": "tianyalan ",
  "name": " tianyalan",
  "email": "tianyalan @email.com"
}
```



而为了让 API 返回数据更具有关联性,我们使用 HAL+JSON 格式,这时候返回的格式就会变成如下形式,这里多了 `_links` 属性,其中有一个 `self.href` 连接指向当前 user 资源。

```
GET http:// api.example.com /users/tianyanlan
Content-Type: application/json
{
  _links: {
    self: {
      href: "/users/tianyanlan"
    }
  }
  "id": "tianyanlan ",
  "name": "tianyanlan",
  "email": "tianyanlan @email.com"
}
```

### (3) 文档服务

HAL 的出现主要弥补普通 JSON 在 API 交互中的不足。让 JSON 更具有描述性,更具有导航性。同时,我们也注意微服务系统是将 API 组合起来为系统提供服务的一种架构风格。在组合 API 时,JSON 这种在格式上缺乏描述性的缺陷就体现得非常明显。我们要为 API 编写文档,要为 API 之间的数据关系和交互方式提供详细的说明。如果我们用 HAL+JSON 描述如下所示的一个带 `location` 属性的 user 信息,我们可以很清楚地知道 user 信息从哪来,location 信息从哪里来,这些信息都在子资源中有所体现。

```
GET http:// api.example.com /users/tianyanlan
Content-Type: application/json
{
  _links: {
    self: {
      href: "/users/tianyanlan"
    }
  }
  "id": "tianyanlan ",
  "name": "tianyanlan",
  "email": "tianyanlan @email.com",
  _embedded: {
    location: {
      _links: {
```



```
self: {  
    href: 'http://api.locationservices.com/locations/1'  
}  
},  
id: 1,  
city: 'hangzhou'  
}  
}
```

使用 HAL 还有一个很大的优势在于其提供的 HAL 浏览器（HAL Browser，<http://api.opensupporter.org/hb2/browser.html#/api/v1>）。HAL 浏览器为我们提供了一种资源可视化的途径。如图 3-21 所示的就是 HAL 浏览器，HAL 所具备的状态、链接和子资源都在该浏览器中有所体现。显然，HAL 浏览器提供了与 Swagger UI 类似的显示效果，可以作为前后台交互的一种动态接口定义文档机制。

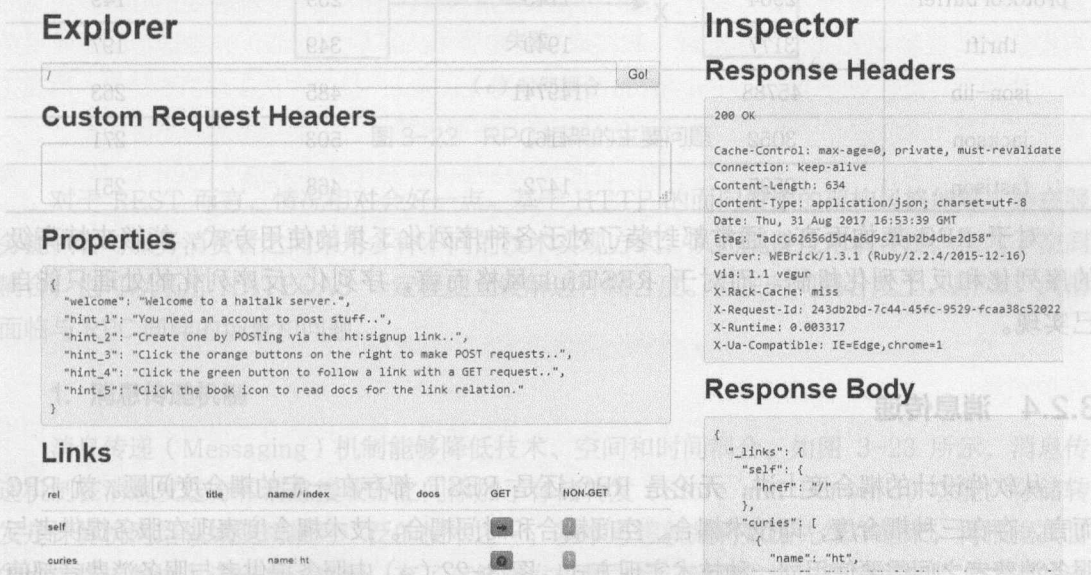


图 3-21 HAL 浏览器（来自 HAL Browser 官网）

### 3. 序列化方式

基于 HTTP 的 REST 能够提供多种不同的响应形式，也可以接受客户端灵活的对资源进行访问的方式。客户端与服务器端的数据交互就涉及序列化问题。所谓序列化（Serialization）就是将对象转化为字节数组，用于网络传输、数据持久化或其他用途，而反



序列化（Deserialization）则是把从网络、磁盘等读取的字节数组还原成原始对象，以便后续业务逻辑操作。

序列化的方式有很多，常见的有文本和二进制两大类。XML 和 JSON 是文本类序列化方式的代表，而二进制实现的方案包括 Google 的 Protocol Buffer 和 Facebook 的 Thrift 等。

性能可能是我们在序列化工具选择过程中最看重的一个指标。性能指标主要包括序列化之后码流大小、序列化/反序列化速度和 CPU/内存资源占用。表 3-2 中我们列举了目前主流的一些序列化技术，可以看到在序列化和反序列化时间维度上 Alibaba 的 fastjson 具有一定优势，而从空间维度上看，相较其他技术我们可以优先选择 Protocol Buffer。

表 3-2 序列化性能比较

	序列化时间	反序列化时间	大 小	压缩后大小
Java	8654	43787	889	541
hessian	6725	10460	501	313
protocol buffer	2964	1745	239	149
thrift	3177	1949	349	197
json-lib	45788	149741	485	263
jackson	3052	4161	503	271
fastjson	2595	1472	468	251

对于 RPC 架构而言，通常都封装了对于各种序列化工具的使用方式，能够支持高级的序列化和反序列化机制。而对于 RESTful 风格而言，序列化/反序列化的处理只能自己实现。

### 3.2.4 消息传递

从软件设计的耦合度上讲，无论是 RPC 还是 REST 都存在一定的耦合度问题。就 RPC 而言，存在三种耦合度，即技术耦合、空间耦合和时间耦合。技术耦合度表现在服务提供者与服务消费者之间需要使用同一种技术实现方式，图 3-22（a）中服务提供者与服务消费者都使用 RMI 作为通信的基本技术，而 RMI 是 Java 领域特有的技术，也就意味着其他服务消费者想要使用该服务也只能采用 Java 作为它的基本开发语言；空间耦合度指的是服务提供者与服务消费者都需要使用统一的方法签名才能相互协作，图 3-22（b）中的 getUserById(id)这个方法名称和参数的定义就是这种耦合的具体体现；而时间耦合度则表现在服务提供者与服务消费者两者只有同时在线才能完成一个完整的服务调用过程，如果出现图 3-22（c）中所示的服务提供者不可用的情况，显然服务消费者调用该服务就会发生失败。



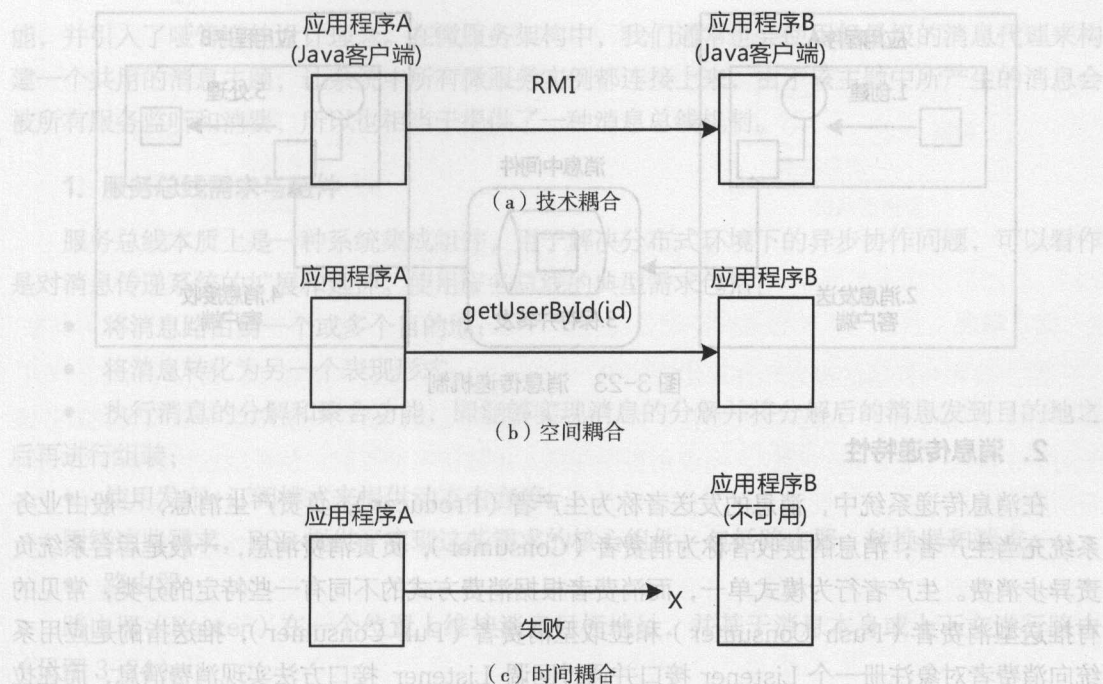


图 3-22 RPC 框架的主要问题

对于 REST 而言，情况相对会好一点。基于 HTTP 的面向资源的架构风格能够支持在服务提供者与服务消费者之间采用多种不同的技术实现方式，从而规避技术耦合度。而对于空间耦合，也可以采用 HATEOAS 一定程度上缓解这种耦合度。但在时间耦合度上，REST 风格面临与 RPC 同样的场景和问题。

## 1. 消息传递机制

消息传递（Messaging）机制能够降低技术、空间和时间耦合。如图 3-23 所示，消息传递机制在消息发送方和消息接收方之间添加了存储转发（Store and Forward）功能。存储转发是计算机网络领域使用最为广泛的技术之一，基本思想就是将数据先缓存起来，再根据其目的地址将该数据发送出去。显然，有了存储转发机制之后，消息发送方和消息接收方之间并不需要相互认识，也不需要同时在线，更加不需要采用同样的实现技术。紧耦合的单阶段方法调用就转变成松耦合的两阶段过程，技术、空间和时间上的约束通过中间层得到显著缓解，这个中间层就是消息传递系统（Messaging System）。



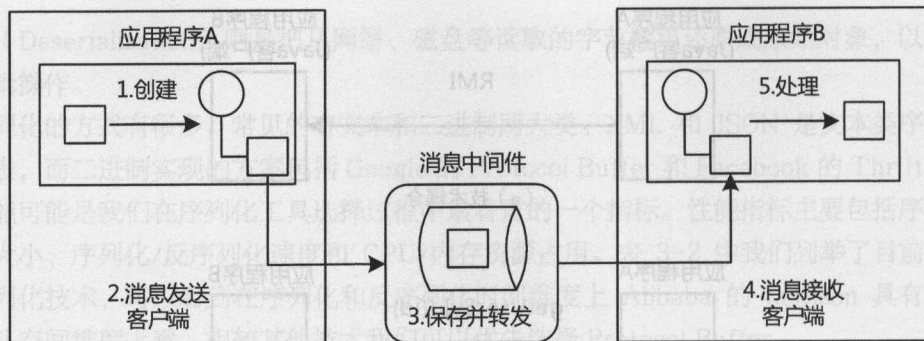


图 3-23 消息传递机制

## 2. 消息传递特性

在消息传递系统中，消息的发送者称为生产者（Producer），负责产生消息，一般由业务系统充当生产者；消息的接收者称为消费者（Consumer），负责消费消息，一般是后台系统负责异步消费。生产者行为模式单一，而消费者根据消费方式的不同有一些特定的分类，常见的有推送型消费者（Push Consumer）和拉取型消费者（Pull Consumer），推送指的是应用系统向消费者对象注册一个 Listener 接口并通过回调 Listener 接口方法实现消费消息，而在拉取方式下应用系统通常主动调用消费者的拉消息方法消费消息，主动权由应用系统控制。

消息传递有两种基本模型，即发布-订阅（Pub-Sub）模型和点对点（Point to Point）模型。发布-订阅支持生产者消费者之间的一对多关系，是典型的推送消费者实现机制；而点对点模型中有且仅有一个消费者，通过实时拉取或基于间隔性拉取的轮询（Polling）方式进行消息消费。在发送者和消费者数量较多的场景下，也可以引入组（Group）的概念，Producer Group 和 Consumer Group 分别代表一类生产者和消费者的集合名称，使用统一逻辑发送和接收消息。

消息持久化是消息传递系统实现存储转发的基本需求，持久化的方式也有多种，可以使用关系型数据库、Key-Value 存储容器和文件系统。持久化存储系统还能实现消息堆积，确保在消息发送高峰期挡住数据洪峰，保证后端系统的稳定性。

在微服务架构中，我们同样需要有一套能够提供消息传递功能的工具和框架，从而实现消息驱动的服务开发和交互能力。

### 3.2.5 服务总线

在本书第 1 章中，我们对比了微服务架构和 ESB 之间的区别。我们知道微服务架构对于 ESB 的改变在于强化端点及弱化通道，抛弃了 ESB 过度复杂的业务规则编排、消息路由等功



能，并引入了哑管道的设计理念。在微服务架构中，我们通常也会使用轻量级的消息代理来构建一个共用的消息主题，让系统中所有微服务实例都连接上来，由于该主题中所产生的消息会被所有服务监听和消费，所以也相当于提供了一种消息总线机制。

1. 服务总线需求与组件

服务总线本质上是一种系统集成组件，用于解决分布式环境下的异步协作问题，可以看作是对消息传递系统的扩展和延伸。使用服务总线的典型需求包括：

- 将消息路由到一个或多个目的地；
- 将消息转化为另一个表现形式；
- 执行消息的分解和聚合功能，即能够实现消息的分解并将分解后的消息发到目的地之后再行组装；
- 使用发布-订阅模式来提供动态内容等。

围绕这些需求，ESB 提供了实现这些需求的核心组件，包括路由器、转换器和端点。

• 路由器

路由器（Router）在一个位置上维护消息目标地址，并基于消息本身或上下文进行路由（见图 3-24）。

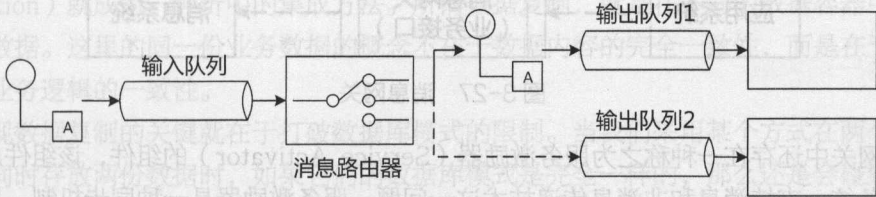


图 3-24 消息路由器结构示意图

• 转换器

转换器（Transformer）用于异构系统之间进行数据适配，数据结构、类型、表现形式、传输方式都是潜在的需要转换的对象（见图 3-25）。

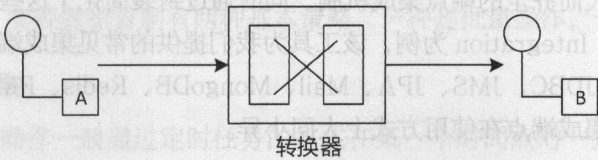


图 3-25 消息转换器结构示意图

• 端点

端点（Endpoint）封装了应用系统与服务总线系统的交互（见图 3-26）。



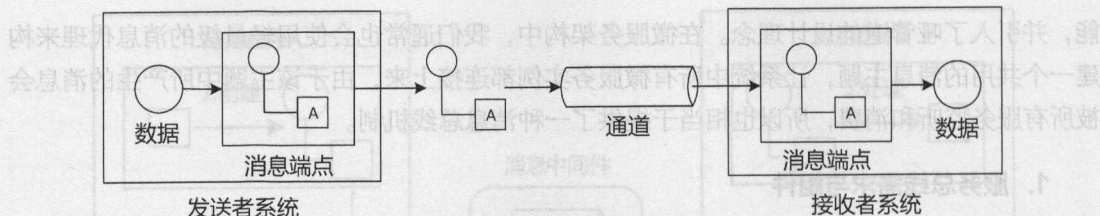


图 3-26 消息端点结构示意图

## 2. 端点

路由器和转换器的概念和作用比较明确，这里对端点再做进一步展开。我们在 1.2.3 节中提到微服务架构对于 ESB 的改变在于强化端点及弱化通道，所以在使用服务总线实现服务集成时，主要关于它的端点机制。当应用程序与消息系统进行交互时，从系统设计的角度讲我们希望应用程序中的业务代码和用于消息传递的非业务代码耦合度尽量低，也就是说应用程序应该封装对消息系统的访问接口。在服务总线中，消息网关（Gateway）就是用来实现这方面的需求。消息网关中应该只包含业务领域层面的接口定义，而不应该出现任何和消息传递技术相关的内容（见图 3-27）。

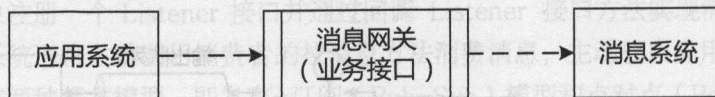


图 3-27 消息网关

消息网关中还存在一种称之为服务激励器（Service Activator）的组件，该组件用于解决如何使服务统一支持消息和非消息传递技术这一问题。服务激励器是一种同步机制，用于屏蔽通信方式，通常用于连接两个通道并把业务代码作用于通过这两个通道的所有消息。

企业服务总线核心组件构成了企业级应用集成的主要实现模式，关于这些实现模式的更多介绍可以参考《企业集成模式》<sup>[25]</sup>。服务总线也可以看作是一种规范，业界基于如何实现服务总线提供了多种第三方工具，如 Mule ESB、Apache Camel 和 Spring Integration。这些工具都为我们提供了强大而齐全的端点集成机制，同时通过封装简化了这些端点的使用方式。以 Spring 家族的 Spring Integration 为例，该工具为我们提供的常见集成端点包括 File、FTP、TCP/UDP、HTTP、JDBC、JMS、JPA、Mail、MongoDB、Redis、RMI、Web Services 等不下数十种，且各个集成端点在使用方式上大同小异。

### 3.2.6 数据复制

共享数据库的集成模式由来已久，在现实的应用系统中，多个系统共享一个数据源的情况



并不少见。从一定程度上讲，开发应用程序的目的就是收集和处理数据，所以数据的生命周期比应用程序更长。尽管共享数据库作为系统集成的代表性模式之一得到广泛应用，但不可否认这种模式存在不少缺点。

## 1. 共享数据库的折中方案

如果共享数据库，数据的存储和表现形式不容易被修改和重构，因为有很多系统对这些数据持有访问权限。一旦对数据做出修改，就可能导致其中一个或多个系统不能正常运作。这就意味着对数据的修改需要协调各个应用系统，这显然会影响到系统的可扩展性。另一方面，这也会导致无法对系统功能进行快速迭代，而业务的快速迭代正是微服务架构所应具有的特性。

对共享数据库最难以把控的一点是如何统一数据库模式（Scheme）。试想如果一个应用系统需要删除某张表中的某个字段，对于普通的场景而言，这无疑是非常简单的事情。但对于共享的数据库而言，由于不知道其他系统是否还在使用该字段，所以也就无法进行直接删除。如果这样的场景很多，那么随着时间的推移，数据的复杂性和可维护性都会对系统造成很多影响。

共享数据库显然不能满足微服务架构中的集成需求，在微服务架构中，我们追求数据的独立性。但对于一些遗留系统而言，我们无法重新打造数据体系，数据复制（Data Replication）就成为一种折中的集成方法。所谓数据复制，就是在不同的数据容器中保存同一份业务数据。这里的同一份业务数据的概念不在于数据内容的完全一致性，而是在于这些数据背后的业务逻辑的一致性。

实现数据复制的关键就在于打破数据库模式的限制。当我们采用某个方式在两个数据存储容器中同时存放两份数据时，如果它们的数据库模式是完全一样的，那么还是会碰到共享数据库模式下的诸多问题。当我们在实施数据复制的集成方式时，将某一份数据库模式转化成其他服务所需要的形式然后再进行数据同步是一项最佳实践。

有了数据复制的设计理念，接下去就要考虑数据冗余所带来的数据一致性（Consistency）的问题。关于数据一致性我们将在第5章中做具体展开，这里只需要明确数据的实时一致性通常都是不需要的，所以可以采取最终一致性（Eventually Consistency）的方式实现数据复制。实现数据复制有两种基本策略，一种是批量操作，一种是事件。

## 2. 实现数据复制

批量（Batch）操作一般通过定时任务的方式在某一个时间点对一批符合复制要求的数据进行同步操作。在实现上，批量操作最好能够支持全量和增量操作，同时为每一批数据确定一个全局唯一的版本号。通过版本号，数据集合就具备选择性和去重性，也就意味着批量操作是一个可以重复执行的过程。批量操作具备一定风险性，由于批量操作本身无法持有状态，利用



版本号把状态放到数据中去。另一方面,在数据的接收方,确保采用一定的数据适配机制实现解耦。采用批量操作实现数据复制的结构图参考图 3-28,这里的数据仓库泛指包含关系型数据库在内的各种数据存储媒介。



图 3-28 批量操作实现数据复制结构图

而对于事件而言,需要将所产生的数据建模成一系列离散事件,我们可以借助于消息传递系统达到数据同步的目的。相比批量操作,事件驱动的数据复制机制能达到较高的数据一致性要求。事件发送方相对简单,只需要将所产生的事件放入事件发布者即可,但对于事件的订阅者而言,可能存在多种表现形式。不管基于何种订阅者模式,在技术实现上我们都可以借助于 3.2.4 节中的消息传递机制达到基于事件的数据复制效果。

### 3.2.7 客户端集成

在本书第 1 章中,我们对微服务架构和 SOA 架构做了对比,其中提到微服务可以直接面向用户,而这里的用户主要针对的就是包括 Web、APP、开放接口等在内的各种客户端程序。关于客户端与微服务之间的集成可以分为以下三种方式。

#### 1. 直接集成

直接集成方式比较简单,就是客户端通过微服务提供的访问入口直接对微服务进行集成(见图 3-29)。这种方式适合于微服务数量不是太多的场景。如果采用直接集成的方式,服务按照业务模块进行边界划分和命名是一项最佳实践。



图 3-29 直接集成方式示意图



## 2. FrontEnd 服务器

FrontEnd 服务器有时候也可以认为是一种 Portal (门户) 机制, 即把客户端所需要的各种 CSS、Javascript 等公共资源统一放在 FrontEnd 服务器, 然后每个微服务包含自身特有的 HTML 等客户端代码片段以及业务逻辑, 通过集成 FrontEnd 服务器上的公共资源完成独立服务的运行。FrontEnd 服务器的结构参考图 3-30。

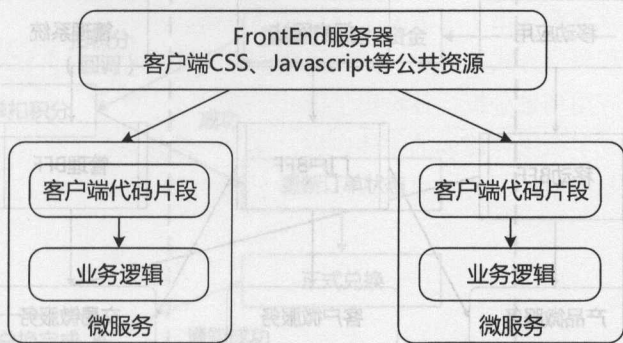


图 3-30 FrontEnd 服务器集成方式示意图

## 3. BackEnd For FrontEnd 服务器

当微服务数量较多且客户端集成场景比较复杂时, 通常就需要单独抽取一层作为客户端访问的统一入口, 这一层在微服务架构里有个专门的叫法称之为 API 网关 (Gateway)。API 网关的基本结构参考图 3-31, 其主要作用是对后端的各个微服务进行整合, 从而为不同的客户端提供定制化的内容。API 网关是微服务架构的基础组件, 具体功能和结构我们将在第 4 章介绍。

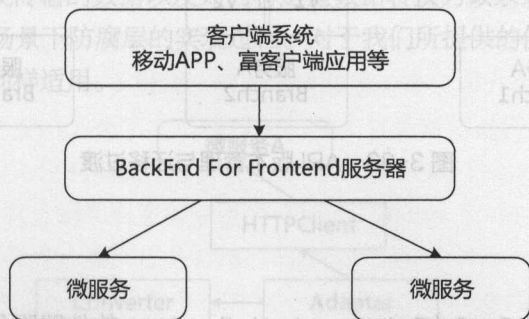


图 3-31 BackEnd For FrontEnd 服务器集成方式示意图

BackEnd For FrontEnd (BFF) 服务器是对 API 网关更为形象的叫法, 也就是专门为前端服务的后端服务器。该服务器在定位上只应该是很薄的一层, 不应该包含任何与业务相关的



逻辑和实现。同时，如果整个系统非常庞大，所有的服务集成都放在一起也会加重这层的维护成本，所以针对不同的业务体系提供专门的 BackEnd For FrontEnd 服务器也是集成过程中的一项最佳实践。图 3-32 展示的就是 BackEnd For FrontEnd 服务器应用的示例，可以看到系统中存在移动后端、门户后端和管理后端三种 BackEnd For FrontEnd 服务器组件，分别面向移动应用、门户网站和内部管理系统。

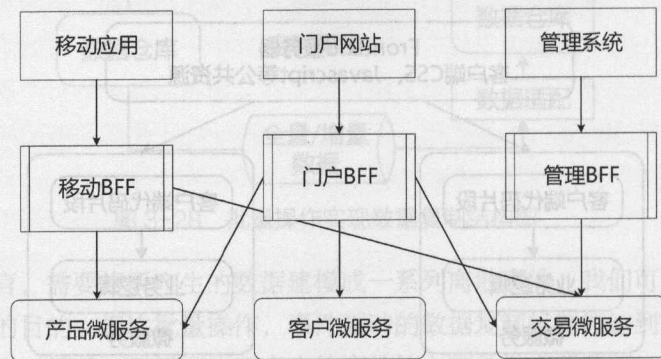


图 3-32 BackEnd For FrontEnd 服务器应用示例

在 BackEnd For FrontEnd 服务器上，由于服务数量大，修改和发布的频率也可能很高，微服务所提供的接口变化在管理上通常采用逐步迁移的方案（见图 3-33）。

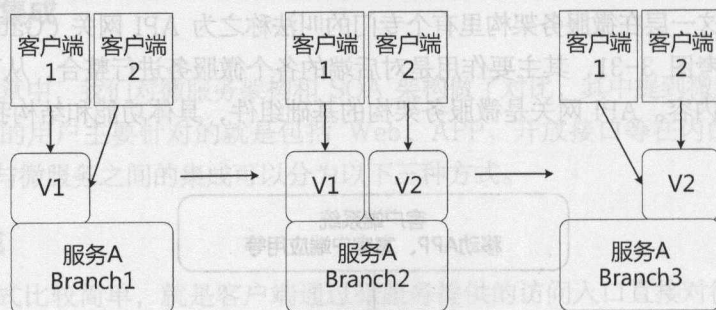


图 3-33 API 版本管理与迁移过渡

### 3.2.8 外部集成

随着服务化思想以及 SaaS（Software As A Service，软件即服务）应用的日渐增多，与外部系统进行集成的方式也发生了很多变化。在服务集成领域，目前基于服务回调的集成方式应用非常广泛。图 3-34 展示的是业务系统与兑吧（<http://www.duiba.com.cn/>）之间的服务交互方式，通过兑吧我们可以把用户在系统中的积分等虚拟货币转化为实物。在实现这一功能



的过程中，我们使用兑吧的服务，而兑吧作为一个第三方的独立服务应该不能依赖于某个具体业务系统，否则势必形成两者之间的强耦合依赖关系。通过回调，兑吧就可以在不需要知道某个具体业务系统的服务实现逻辑的前提下完成整个积分兑换工作。在图 3-34 中的扣积分和通知成功两步操作实际上都是通过回调实现。

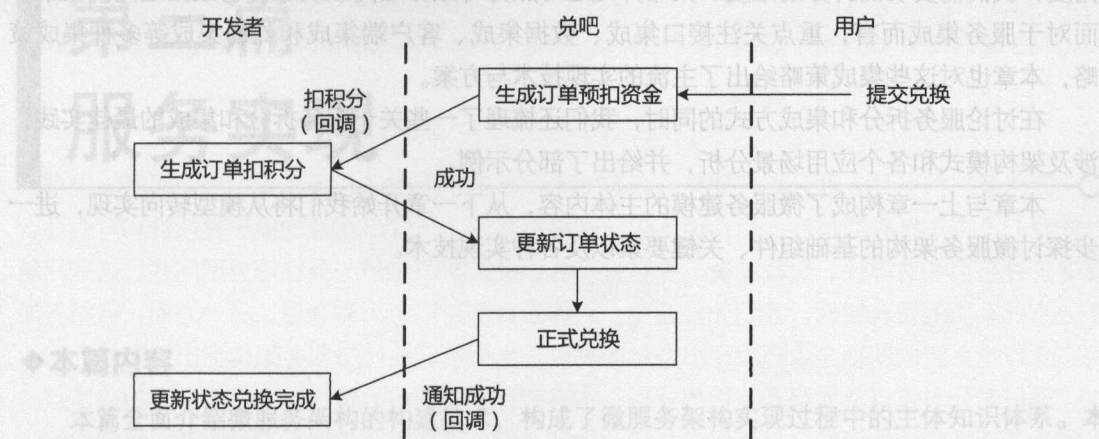


图 3-34 兑吧兑换流程中的回调关系

对于图 3-34 中所示的集成方式，实际上也可以简单理解为服务与服务的集成，只不过有些服务是来自第三方平台。回调作为消除循环依赖的一种有效方式，只需要我们提供回调入口即可完成与外部系统的集成。整个服务交互过程中，在服务访问入口添加防腐层是一项最佳实践。而防腐层的建立通常需要实现适配（Adapt）和转换（Convert）。考虑这样一个场景，当外部系统通过基于 RESTful 风格暴露访问接口给我们，我们在使用该服务时，就需要考虑如何获取通过 HTTP 协议传输的数据以及如何将这些数据转换为该系统自身所能识别的业务数据。图 3-35 展示了该场景下防腐层的实现过程。对于我们所提供的供外部系统访问的回调接口，防腐层的设计理念同样适用。

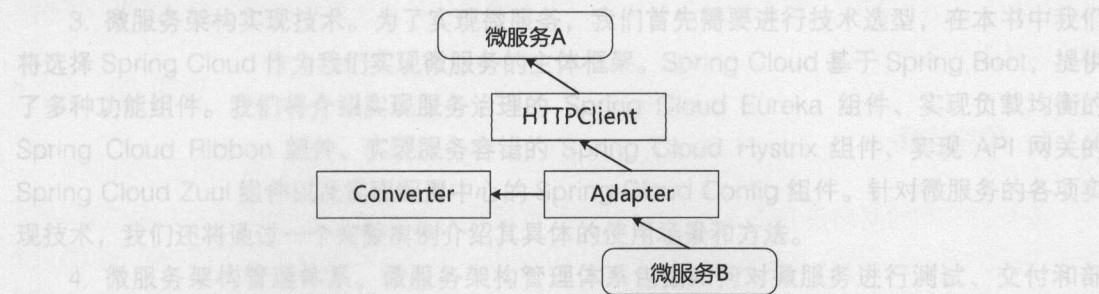


图 3-35 防腐层示例



## 3.3 本章小结

服务拆分和集成构成了从单块系统向微服务系统转变过程中最基本的切入点。从服务拆分角度，我们需要明确拆分的维度、策略并处理好服务与服务之间以及服务与数据之间的关系。而对于服务集成而言，重点关注接口集成、数据集成、客户端集成和外部集成等多种集成策略，本章也对这些集成策略给出了主流的实现技术与方案。

在讨论服务拆分和集成方式的同时，我们还梳理了一些关于服务拆分和集成的最佳实践，涉及架构模式和各个应用场景分析，并给出了部分示例。

本章与上一章构成了微服务建模的主体内容，从下一章开始我们将从模型转向实现，进一步探讨微服务架构的基础组件、关键要素以及各种实现技术。

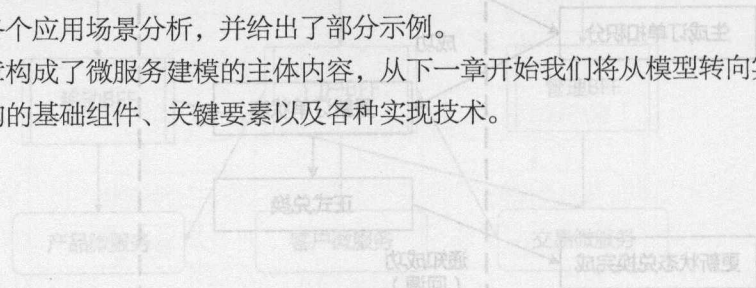


图 3-32 BackEnd For FrontEnd 服务应用示例

图 3-33 基于 HTTP 的集成方式



图 3-33 基于 HTTP 的集成方式

### 3.2.8 外部集成

随着服务化思想以及 SaaS (Software as a Service) 软件即服务应用的日渐增多，与外部系统进行集成的方式也发生了变化。在服务集成领域，目前基于服务回调的集成方式应用非常广泛。图 3-34 展示的是业务系统与兑吧 (http://www.duiha.com.cn/) 之间的服务交互方式。通过兑吧我们可以把用户在系统中的积分等虚拟货币转化为实物。在实现这一功能



## 第三篇 服务实现

### ◆ 本篇内容

本篇全面介绍微服务架构的构建技术，构成了微服务架构实现过程中的主体知识体系。本篇分成四章，分别从四个主要方面阐述微服务实现的不同维度，采用从原理分析到技术实现的行文思路。

1. 微服务架构基础组件。微服务架构的实现首先需要提供一系列基础组件，包括服务与服务之间的通信、事件驱动架构、集群与负载均衡、服务路由等分布式环境下的通用组件，也包括 API 网关和配置管理等微服务架构所特有的功能组件。

2. 微服务架构关键要素。在微服务架构中，我们还要考虑一些实现上的关键要素。基于服务注册中心的服务发布和订阅机制是微服务体系下实现服务治理的基本手段；在分布式环境下追求最终一致性的可靠事件模式、补偿模式、Sagas 长事务模式、TCC 模式、最大努力通知模式等各种数据一致性模式的实现方式也值得我们深入分析。同时，关于如何保证服务的可靠性，我们也会对服务容错、服务隔离、服务限流和服务降级等方面进行展开。

3. 微服务架构实现技术。为了实现微服务，我们首先需要进行技术选型，在本书中我们将选择 Spring Cloud 作为我们实现微服务的主体框架。Spring Cloud 基于 Spring Boot，提供了多种功能组件。我们将介绍实现服务治理的 Spring Cloud Eureka 组件、实现负载均衡的 Spring Cloud Ribbon 组件、实现服务容错的 Spring Cloud Hystrix 组件、实现 API 网关的 Spring Cloud Zuul 组件以及实现配置中心的 Spring Cloud Config 组件。针对微服务的各项实现技术，我们还将通过一个完整案例介绍其具体的使用场景和方法。

4. 微服务架构管理体系。微服务架构管理体系包括如何对微服务进行测试、交付和部署，也包括基于日志聚合和服务跟踪的服务监控管理。同时，服务安全对于微服务架构也提供了不同的实现策略和技术。



## 第 4 章

# 微服务架构基础组件

略，本章也对这些集成策略给出了主流的实现技术与方案。

在讨论服务拆分和集成方式的同时，我们还梳理了一些关于微服务架构的常见问题。

微服务架构也是一种分布式架构，分布式系统相较于集中式系统具备优势的同时，也存在一些我们不得不考虑的特性，包括但不限于网络传输的多态性、服务的维护复杂性和可用性、服务的分布性和对等性、负载均衡、分布式事务以及数据节点的一致性等问题。这些问题是分布式系统的固有特性，我们无法避免，只能想办法进行利用和管理。另一方面，微服务架构在分布式系统的基础之上还表现出特定的需求，也给我们设计微服务架构提出了挑战。

通过对分布式以及微服务特性的分析，本书把微服务架构的设计归为两类主要的关注点。一类是基础性的功能组件，包括服务之间的通信、面向事件驱动的架构设计方法、负载均衡、服务路由、API 网关和分布式配置中心等；另一类则是微服务架构实现上的关键要素，更多关注于服务之间调用的稳定性、可用性以及高效性，包括服务的治理、分布式环境下的数据一致性以及服务的可靠性设计。

本章将主要围绕实现微服务架构的基础性功能组件展开讨论，而微服务架构实现上的关键要素我们将在下一章中具体介绍。

### 4.1 服务通信

网络通信是任何分布式系统的基础组件。网络通信本身涉及面很广，这里并不打算介绍网络通信相关的方方面面。对于微服务架构而言，网络通信关注于网络连接、IO 模型、可靠性设计以及服务调用方式。

#### 4.1.1 网络连接

基于 TCP 的网络连接有两种基本方式，也就是通常所说的长连接（也叫持久连接，Persistent Connection）和短连接（Short Connection）。当网络通信采用 TCP 时，在真正的读写操作之前，Server 与 Client 之间必须建立一个连接，当读写操作完成后，双方不再需要这个连接时就可以释放这个连接。连接的建立需要三次握手 [见图 4-1 (a)]，而释放则需要



四次握手 [见图 4-1 (b)], 每个连接的建立都意味着需要资源和时间的消耗。

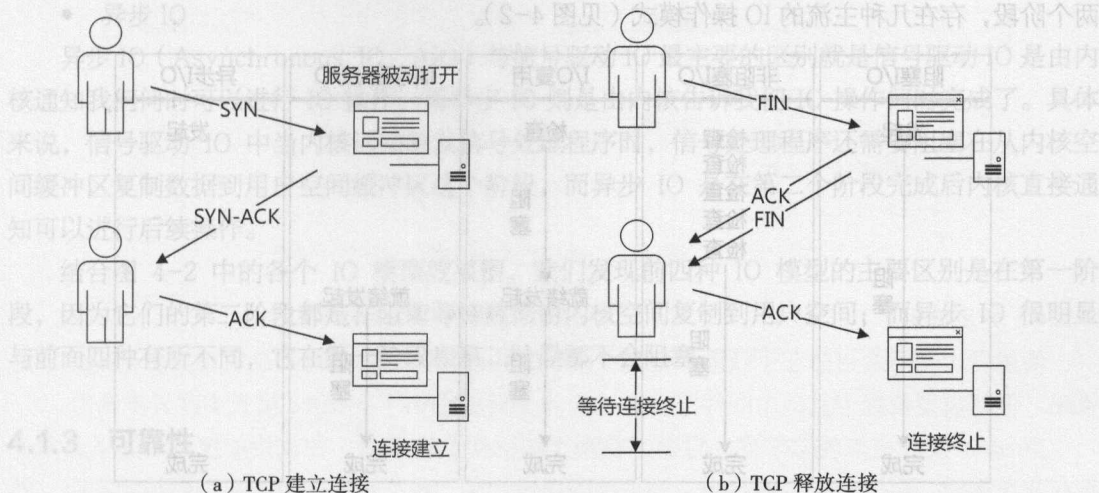


图 4-1 TCP 握手协议

当客户端向服务器端发起连接请求，服务器端接收请求，然后双方就可以建立连接。服务器端响应来自客户端的请求就需要完成一次读写过程，这时候双方都可以发起关闭操作。所以短连接一般只会在客户端/服务器端间传递一次读写操作，也就是说 TCP 连接建立后，数据包传输完成即关闭连接。短连接结构简单，管理起来比较简单，存在的连接都是有用的连接，不需要额外的控制手段。

长连接则不同，当客户端与服务器端完成一次读写之后，它们之间的连接并不会主动关闭，后续的读写操作会继续使用这个连接。这样当 TCP 连接建立后，就可以连续发送多个数据包，能够节省资源并降低时延。

长连接和短连接的产生在于客户端和服务器端采取的关闭策略，具体的应用场景采用具体的策略，没有十全十美的选择，只有合适的选择。

## 4.1.2 IO 模型

说到网络通信，就不得不提 IO 模型。现代操作系统都包括内核空间（Kernel Space）和用户空间（User Space），内核空间主要存放内核代码和数据，是供系统进程使用的空间；而用户空间主要存放的是用户代码和数据，是供用户进程使用的空间。一般的 IO 操作都分为两个阶段，以网络套接口（Socket）的输入操作为例，它的两个阶段包括内核空间和用户空间之间的数据传输，即首先等待网络数据到来，当数据分组到来时，将其拷贝到内核空间的临时缓



缓冲区中，然后再将内核空间临时缓冲区中的数据复制到用户空间缓冲区中。围绕 IO 操作的这两个阶段，存在几种主流的 IO 操作模式（见图 4-2）。

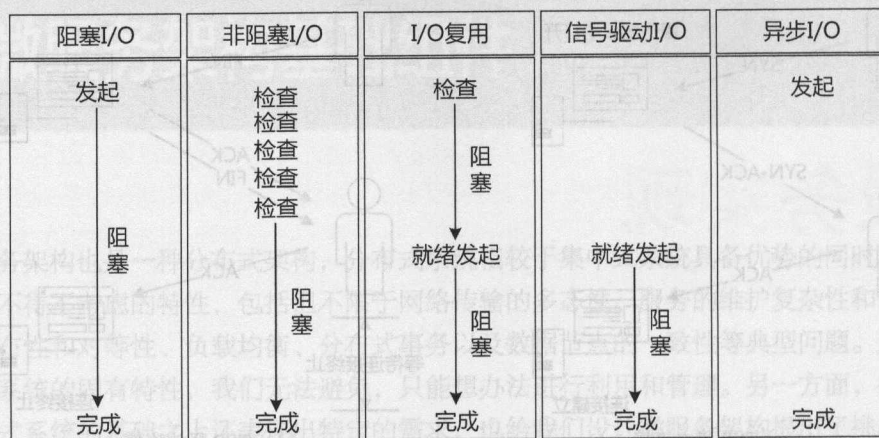


图 4-2 操作系统 IO 模型

图 4-2 中每个模式对应的不同的处理方式和效果如下。

- 阻塞 IO

阻塞 IO（Blocking IO，BIO）在默认情况下，所有套接口都是阻塞的，意味着 IO 的发起和结束都需等待。任何一个系统调用都会产生一个由用户态到内核态切换，再从内核态到用户态切换的过程，而进程上下文切换是通过系统中断程序来实现的，需要保存当前进程的上下文状态，这是一个成本很高的过程。

- 非阻塞 IO

如果采用非阻塞 IO（Non-blocking IO，NIO），即当我们把套接口设置成非阻塞时，会由用户进程不停地询问内核某种操作是否准备就绪，这就是我们常说的轮询（Polling）。这同样是一件比较浪费 CPU 的方式。

- IO 复用

IO 复用主要依赖于操作系统提供的 select 和 poll 机制。同样会阻塞进程，但是这里进程是阻塞在 select 或者 poll 这两个系统调用上，而不是阻塞在真正的 IO 操作上。另外还有一点不同于阻塞 IO 的就是，尽管看起来 IO 复用阻塞了两次，但是第一次阻塞是在 select 上，select 可以监控多个套接口上是否已有 IO 操作准备就绪，而不是像阻塞 IO 那种，一次只能监控一个套接口。

- 信号驱动 IO

信号驱动 IO 就是说我们可以通过 sigaction 系统调用注册一个信号处理程序，然后主程序可以继续向下执行，当我们所监控的套接口有 IO 操作准备就绪时，由内核通知触发前面注册



的信号处理程序执行，然后将我们所需要的数据从内核空间复制到用户空间。

- 异步 IO

异步 IO（Asynchronous IO，AIO）与信号驱动 IO 最主要的区别就是信号驱动 IO 是由内核通知我们何时可以进行 IO 操作，而异步 IO 则是由内核告诉我们 IO 操作何时完成了。具体来说，信号驱动 IO 中当内核通知触发信号处理程序时，信号处理程序还需要阻塞在从内核空间缓冲区复制数据到用户空间缓冲区这个阶段，而异步 IO 是在第二个阶段完成后内核直接通知可以进行后续操作。

结合图 4-2 中的各个 IO 模型效果图，我们发现前四种 IO 模型的主要区别是在第一阶段，因为它们的第二阶段都是在阻塞等待数据由内核空间复制到用户空间；而异步 IO 很明显与前面四种有所不同，它在第一阶段和第二阶段都不会阻塞。

### 4.1.3 可靠性

由于存在网络闪断、超时等网络状态相关的不稳定性以及业务系统本身的故障，网络之间的通信必须在发生上述问题时能够快速感知并修复。常见的网络通信保障手段包括链路有效性检测以及断线之后的重连处理。

从原理上讲，要确保通信链路的可靠性就必须对链路进行周期性的有效性检测，通用的做法就是心跳（Heart Beat）检测。心跳检测通常有两种技术实现方式，一种是在 TCP 层通过建立长连接在发送方和接收方之间传递心跳信息；另一种则是在应用层，心跳信息根据系统要求可能包含一定的业务逻辑。

当发送方检测到通信链路中断，会在事先约定好的重连间隔时间之后发起重连操作，如果重连失败，则周期性地使用该间隔时间进行重连直至重连成功。

### 4.1.4 同步与异步

服务通信存在两种基本方式（见图 4-3），即单向（One Way）模式和请求应答（Request-Response）模式，前者体现为异步操作，而后者一般执行同步操作。

- 同步调用

同步调用会造成业务线程阻塞，但开发和管理相对简单。同步调用时序图参考图 4-3，我们可以看到服务线程发送请求到 IO 线程之后就一直处于等待阶段，直到 IO 线程完成与网络的读写操作之后被主动唤醒。



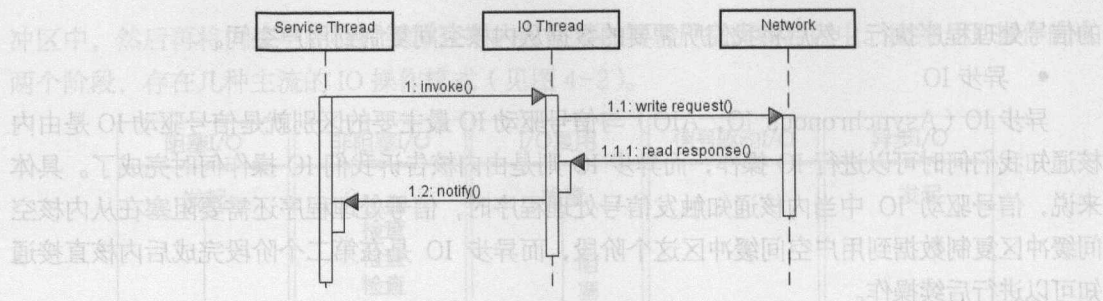


图 4-3 同步调用时序图

### 异步调用

使用异步调用的目的在于获取高性能，队列思想和事件驱动架构都是实现异步调用的常见策略，但都需要依赖于基础中间件平台。JDK 也为我们提供了 Future 模式实现异步调用。

Future 模式有点类似于商品订单，在网上购物提交订单后，在收货的这段时间里无需一直在家里等候，可以先干别的事情。类推到程序设计中，提交请求的目的是期望得到响应，但这个响应可能很慢。传统做法是一直等待到这个响应收到后再去做别的事情，但如果利用 Future 模式就无需等待响应的到来，在等待响应的过程中可以执行其他程序。传统调用和 Future 模式调用对比可以参考图 4-4，我们可以看到在 Future 模式调用过程中，服务调用者向服务提供者发出请求后马上返回，可以继续执行其他任务直到服务提供者通知 Future 调用的结果，体现了 Future 调用异步化特点。

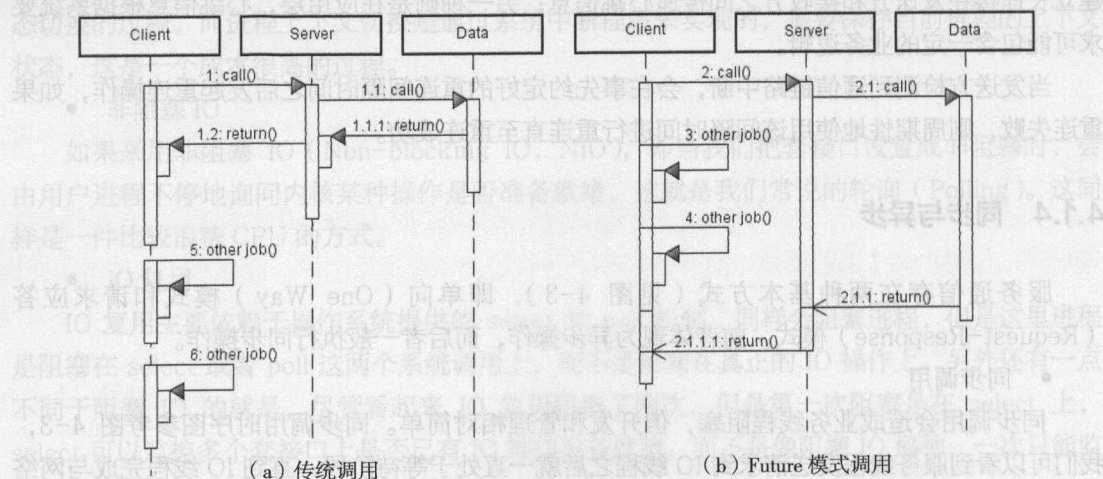


图 4-4 Java Future 机制

Future 调用可以进一步细分成两种模式，Future-Get 模式和 Future-Listener 模式。Future-Get 模式参考图 4-5，可以看到这种模式下通过主动 get 结果的方式获取 Future 结



果, 而这个 get 过程是串行的, 会造成执行 get 方法的线程形成阻塞。

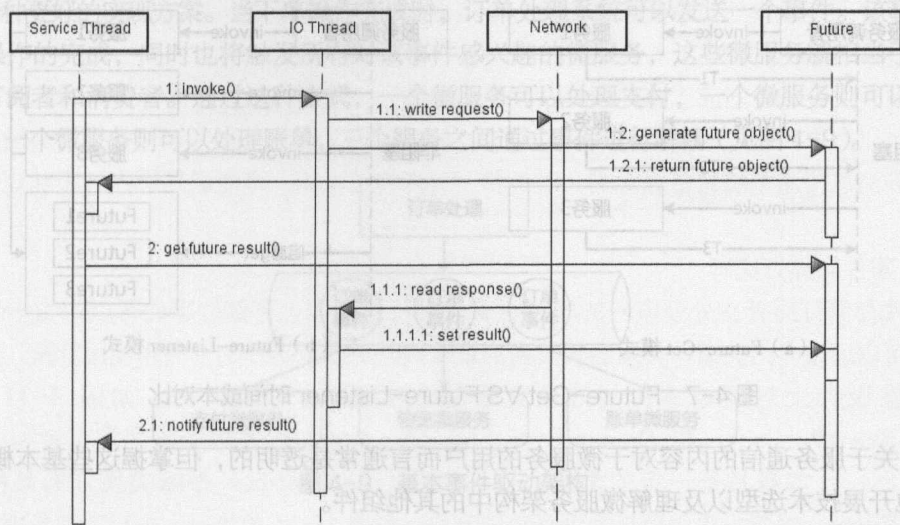


图 4-5 Future-Get 串行模式时序图

而 Future-Listener 模式则不同(参考图 4-6), 在 Future-Listener 模式中需要创建 Listener, 当 Future 结果生成时会唤醒注册到该 Future 上的 Listener 对象, 从而形成异步回调机制。

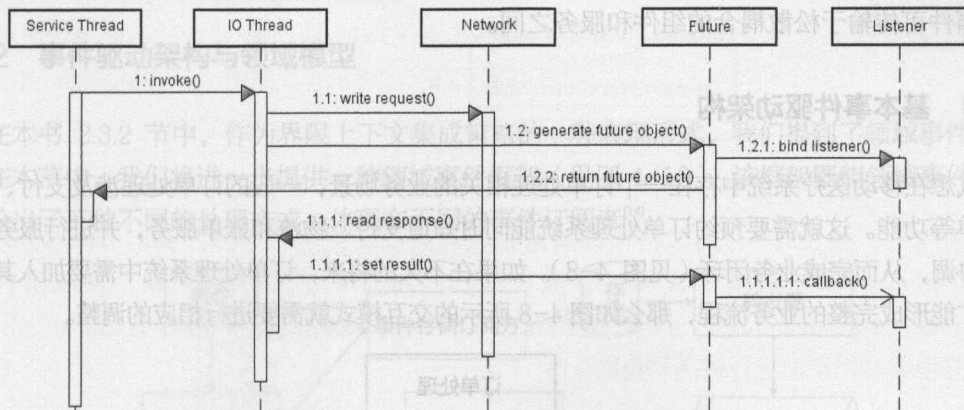


图 4-6 Future-Listener 并行模式

Future-Get 和 Future-Listener 两种模式的对比可以参考图 4-7。假设有三个任务的执行时间分别是  $T_1$ 、 $T_2$  和  $T_3$ , 则对于 Future-Get 而言执行这三个任务的总时间为  $T=T_1+T_2+T_3$ , 而 Future-Listener 中  $T=\text{Max}(T_1, T_2, T_3)$ , 显然 Future-Listener 的



时间成本小于 Future-Get。

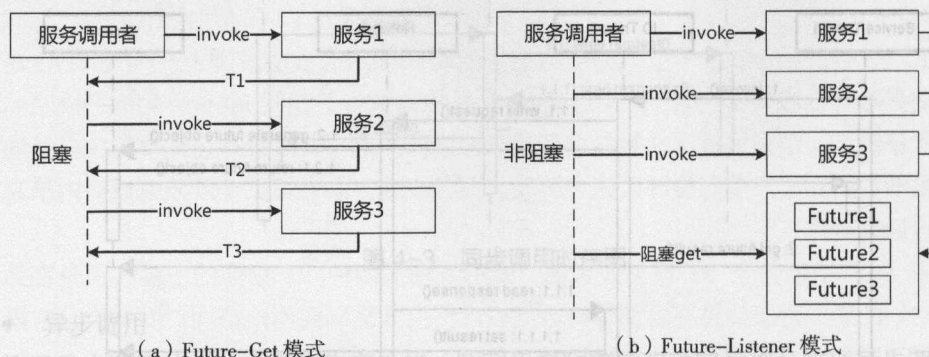


图 4-7 Future-Get VS Future-Listener 时间成本对比

以上关于服务通信的内容对于微服务的用户而言通常是透明的，但掌握这些基本概念有助于更好地开展技术选型以及理解微服务架构中的其他组件。

## 4.2 事件驱动

在本节中，我们将讨论事件驱动架构（Event-Driven Architecture, EDA）及其在微服务架构中的应用。事件驱动架构定义了一个设计和实现应用系统的架构风格，在这个架构风格里，事件可传输于松散耦合的组件和服务之间。

### 4.2.1 基本事件驱动架构

试想移动医疗系统中存在一个订单处理相关的业务场景，一般的订单处理涉及支付、物流和账单等功能。这就需要预约订单处理系统能同时知道支付、物流和账单服务，并进行服务的调用和协调，从而完成业务闭环（见图 4-8）。如果在不久的将来，订单处理系统中需要加入其他的服

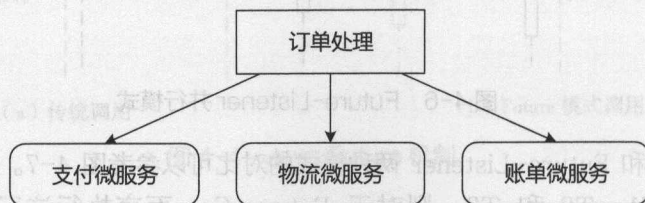


图 4-8 基本的服务交互模式



考虑到系统扩展性，显然图 4-8 中的服务交互模式并不是一个好的选择，事件驱动架构为我们提供了一种更好的实现方案。当下单操作完成时，订单处理系统可以发送一个事件。该事件表明了这个操作的完成，同时也将触发所有对该事件感兴趣的微服务，这些微服务就相当于这个事件的订阅者和消费者。通过这种方式，一个微服务可以处理支付，一个微服务则可以处理物流，而另一个微服务则可以处理账单，三个服务之间通过事件进行解耦（见图 4-9）。

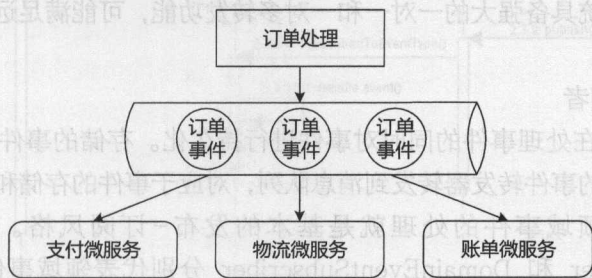


图 4-9 基本事件驱动架构

在图 4-9 中，基本事件处理架构的优势就在于当系统中需要添加另一个业务逻辑来完成整个流程时，只需要对该事件添加一个订阅者即可，不需要对订单处理系统做任何修改。考虑到在微服务架构中，服务数量较多且不可避免需要对服务进行重构，事件处理架构在系统扩展性上优势就尤为明显。而在技术实现上，通过消息传递机制，我们不必花费太大代价就能实现事件驱动架构。

## 4.2.2 事件驱动架构与领域模型

在本书 2.3.2 节中，作为界限上下文集成策略的一种表现形式，我们提到了领域事件的概念。在本节中，我们将进一步提供一种领域事件框架（见图 4-10），该框架围绕领域事件生命周期给出了三种不同的处理方式，体现在不同的事件订阅者<sup>[4]</sup>。

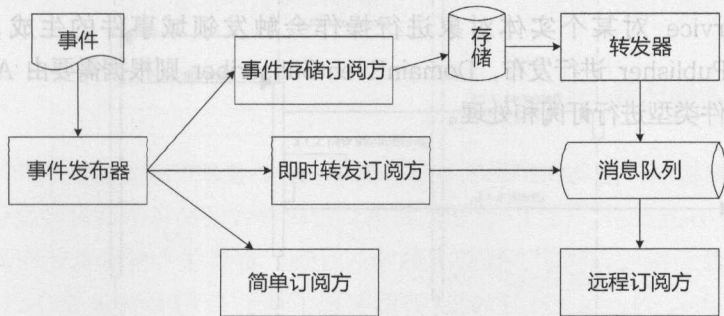


图 4-10 事件驱动架构



- 简单订阅者

简单订阅者直接处理事件，表现为一个独立的事件处理程序，对应于事件的使用阶段。

- 即时转发订阅者

即时转发订阅者对应于事件的分发和使用阶段，一方面可以具备简单订阅者的功能，另一方面也可以把事件转发给其他订阅者。通常，把事件转发到消息队列是一个好的实践方法，现有的很多消息传递系统具备强大的一对一和一对多转发功能，可能满足远程订阅者处理事件的需求。

- 事件存储订阅者

事件存储订阅者在处理事件的同时对事件进行持久化。存储的事件可以作为一种历史记录，也可以通过专门的事件转发器转发到消息队列，对应于事件的存储和使用阶段。

架构设计上对领域事件的处理就是基本的发布-订阅风格。如图 4-11 所示，DomainEventPublisher 和 DomainEventSubscriber 分别代表领域事件发布者和订阅者，DomainEvent 本身具备一定的类型，DomainEventSubscriber 根据类型订阅某种特定的 DomainEvent。

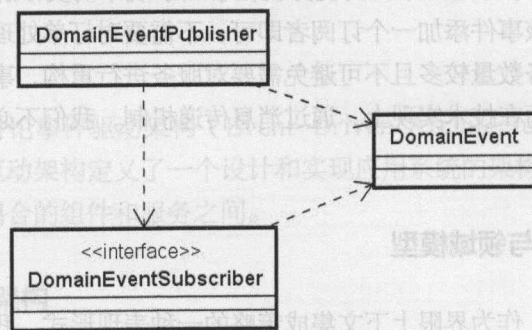


图 4-11 事件的发布与订阅

图 4-12 是发布-订阅风格中涉及的领域对象以及交互时序图，我们可以看到应用层 ApplicationService 对某个实体对象进行操作会触发领域事件的生成，领域事件通过 DomainEventPublisher 进行发布，DomainEventSubscriber 则根据需要由 ApplicationService 创建并根据事件类型进行订阅和处理。

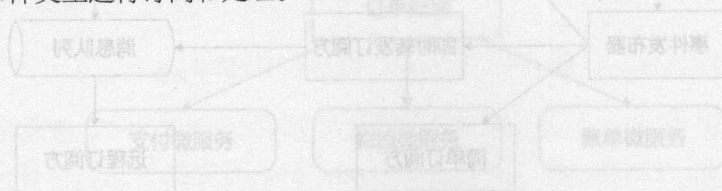


图 4-12 基本的发布-订阅风格



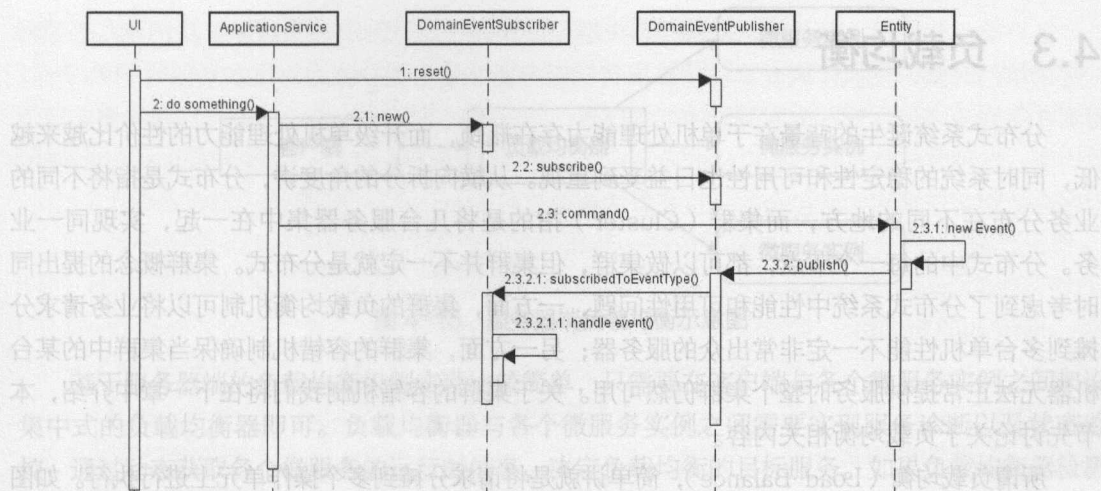


图 4-12 发布订阅时序图

领域事件既可以由本地界限上下文消费，也可以由远程的界限上下文消费。远程界限上下文发布领域事件需要考虑消息的最终一致性、同步和异步以及领域事件存储等问题。尤其针对远程交互本身存在的网络稳定性等各种不可控原因，一般都会对事件进行存储以便发生问题时跟踪和重试。支持不同事件类型、支持领域事件和存储事件之间的转换、检索由领域模型所产生的所有结果的历史记录、使用事件存储中的数据进行业务预测和分析是常见的事件存储需求。事件可以通过 DomainEventPublisher 进行集中式的存储，也可以分别保存在各个 DomainEvent Subscriber 中。图 4-13 就是一个包含事件存储的发布订阅时序图，通过 DomainEventSubscriber 构建 EventStore 进行事件存储。

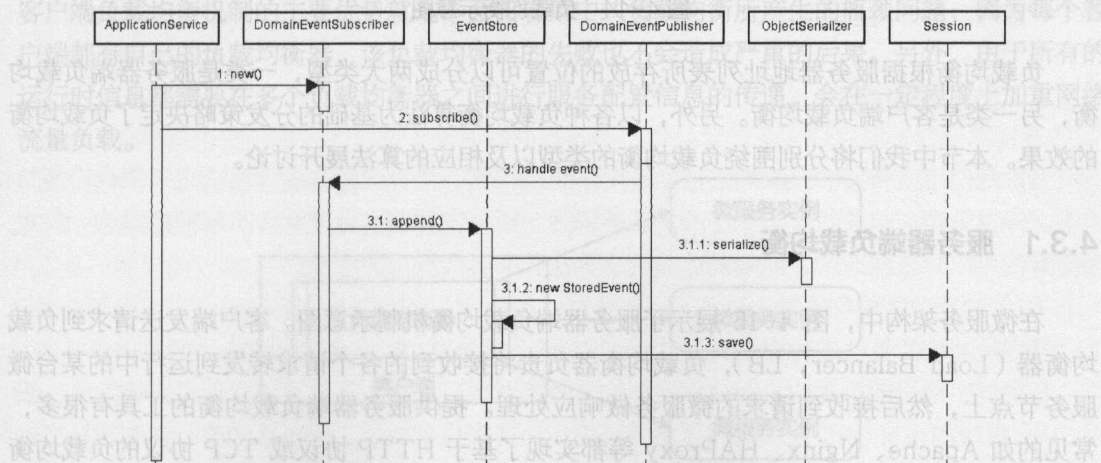


图 4-13 包含事件存储的发布订阅时序图



## 4.3 负载均衡

分布式系统诞生的背景在于单机处理能力存在瓶颈，而升级单机处理能力的性价比越来越低，同时系统的稳定性和可用性也日益受到重视。从横向拆分的角度讲，分布式是指将不同的业务分布在不同的地方，而集群（Cluster）指的是将几台服务器集中在一起，实现同一业务。分布式中的每一个节点，都可以做集群，但集群并不一定就是分布式。集群概念的提出同时考虑到了分布式系统中性能和可用性问题，一方面，集群的负载均衡机制可以将业务请求分摊到多台单机性能不一定非常出众的服务器；另一方面，集群的容错机制确保当集群中的某台机器无法正常提供服务时整个集群仍然可用。关于集群的容错机制我们将在下一章中介绍，本节先讨论关于负载均衡相关内容。

所谓负载均衡（Load Balance），简单讲就是将请求分摊到多个操作单元上进行执行。如图 4-14 所示，来自客户端的请求通过负载均衡机制将被分发到各个服务器，根据分发策略的不同将产生对应的分发结果。负载均衡建立在现有网络结构之上，它提供了一种廉价、有效、透明的方法扩展服务器的带宽、增加吞吐量、加强网络数据处理能力，以及提高网络的灵活性。负载均衡在实现上可以使用硬件、软件或者两者兼有，本书主要介绍基于软件的负载均衡机制。

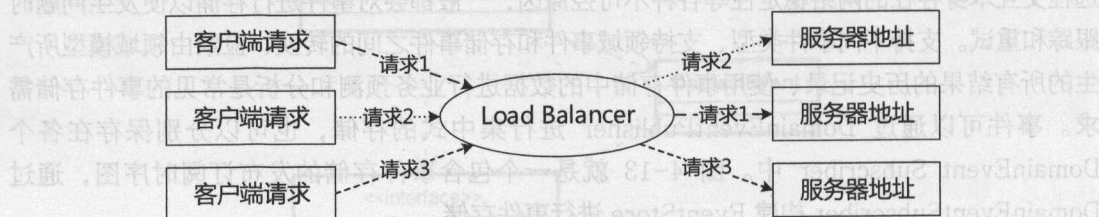


图 4-14 负载均衡示意图

负载均衡根据服务器地址列表所存放的位置可以分成两大类型，一类是服务器端负载均衡，另一类是客户端负载均衡。另外，以各种负载均衡算法为基础的分发策略决定了负载均衡的效果。本节中我们将分别围绕负载均衡的类型以及相应的算法展开讨论。

### 4.3.1 服务器端负载均衡

在微服务架构中，图 4-15 展示了服务器端负载均衡机制示意图。客户端发送请求到负载均衡器（Load Balancer，LB），负载均衡器负责将接收到的各个请求转发到运行中的某台微服务节点上，然后接收到请求的微服务做响应处理。提供服务器端负载均衡的工具有很多，常见的如 Apache、Nginx、HAProxy 等都实现了基于 HTTP 协议或 TCP 协议的负载均衡模块。



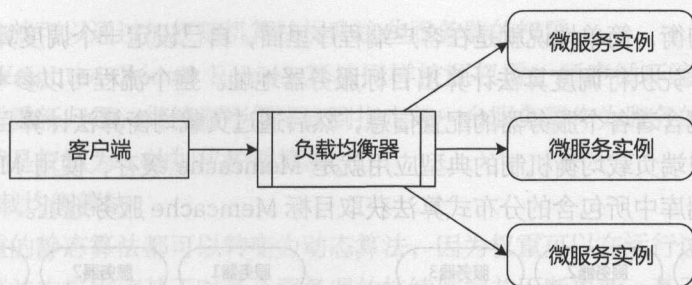


图 4-15 服务器端负载均衡示意图

基于服务器端的负载均衡机制实现比较简单，只需要在客户端与各个微服务实例之间架设集中式的负载均衡器即可。负载均衡器与各个微服务实例之间需要实现服务诊断以及状态监控，通过动态获取各个微服务的运行时信息，决定负载均衡的目标服务。如果负载均衡器检测到某个服务已经不可用时就会自动移除该服务。

通过上述分析，可以看到负载均衡器运行在一台独立的服务器上并充当代理（Proxy）的作用。所有的请求都需要通过负载均衡器的转发才能实现服务调用，这可能会是一个问题，因为当服务请求量越来越大时，负载均衡器将会成为系统的瓶颈。同时，一旦负载均衡器自身发生失败，整个微服务的调用过程都将发生失败。因此，在微服务架构中，为了避免集中式负载均衡所带来的这种问题，客户端负载均衡同样也是一种常用的方式。

### 4.3.2 客户端负载均衡

客户端本身同样可以实现负载均衡，客户端负载均衡最基本的表现形式如图 4-16 所示。客户端负载均衡机制的主要优势就是不会出现集中式负载均衡所产生的瓶颈问题，因为每个客户端都有自己的负载均衡器，该负载均衡器的失败也不会造成严重的后果。另外，由于所有的运行时信息都需要在多个负载均衡器之间进行服务配置信息的传递，会在一定程度上加重网络流量负载。

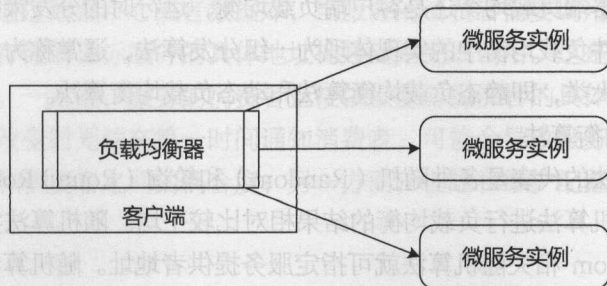


图 4-16 客户端负载均衡示意图



客户端负载均衡, 简单地讲就是在客户端程序里面, 自己设定一个调度算法, 在向服务器发起请求的时候, 先执行调度算法计算出目标服务器地址。整个流程可以参考图 4-17, 可以看到在微服务中包含着各个服务器的配置信息, 然后通过负载均衡算法计算目标服务器实现负载均衡。这种客户端负载均衡机制的典型应用就是 Memcache 缓存, 使用 Memcache 的应用程序会依赖客户端库中所包含的分布式算法获取目标 Memcache 服务地址。

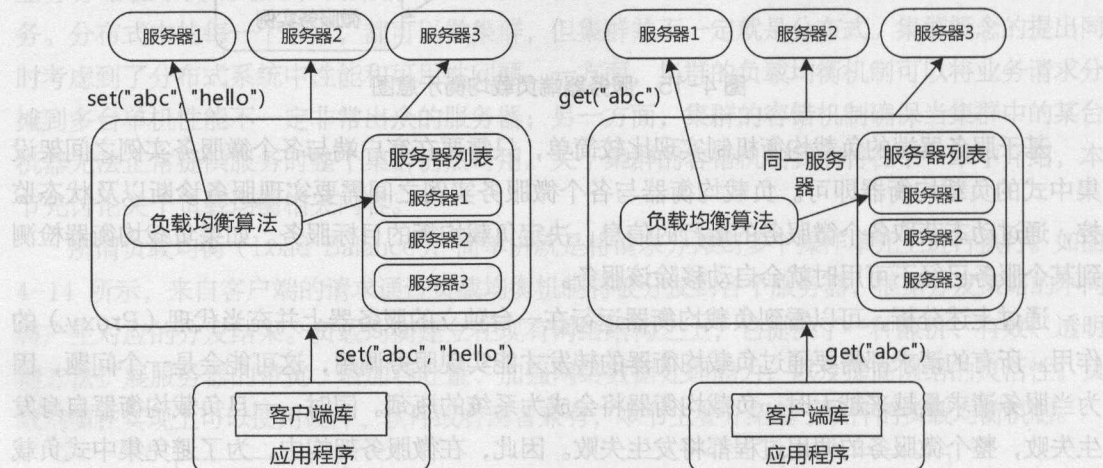


图 4-17 客户端实现负载均衡算法示意图

客户端负载均衡的另一种典型实现方式是把 Nginx 等能够实现代理功能的负载均衡器部署到运行微服务的同一台机器上。当然, 这种方式需要考虑实施成本和维护性问题。

客户端负载均衡比较适合于客户端具有成熟的调度库函数、算法以及 API 的工具和框架。一般可以选择为初期简单的负载均衡方案, 也可以结合其他负载均衡方案进行架构。

### 4.3.3 负载均衡算法

无论是使用服务器端负载均衡还是客户端负载均衡, 运行时的分发策略决定了负载均衡的效果。分发策略在软件负载均衡中的实现体现为一组分发算法, 通常称为负载均衡算法。负载均衡算法可以分成两大类, 即静态负载均衡算法和动态负载均衡算法。

#### (1) 静态负载均衡算法

静态负载均衡算法的代表是各种随机 (Random) 和轮询 (Round Robin) 算法。

在集群中采用随机算法进行负载均衡的结果相对比较平均。随机算法实现也比较简单, 使用 JDK 自带的 Random 相关随机算法就可指定服务提供者地址。随机算法的一种改进是加权随机 (Weight Random) 算法, 在集群中可能存在部分性能较优的服务器, 为了使这些服务



器响应更多请求，就可以通过加权随机算法提升这些服务器的权重。

加权轮循（Weighted Round Robin）算法同样按照权重，顺序循环遍历服务提供者列表，到达上限之后重新归零，继续顺序循环直到指定某一台服务器作为服务的提供者。普通的轮询算法实际上就是权重为 1 的加权轮循算法。

### （2）动态负载均衡算法

所有涉及权重的静态算法都可以转变为动态算法，因为权重可以在运行过程中动态更新。例如，动态轮询算法中权重值基于对各个服务器的持续监控并不断更新。基于服务器的实时性能分析分配连接（如每个节点的当前连接数或者节点的最快响应时间）是常见的动态策略。类似的动态算法还包括最少连接数（Least Connection）算法和服务调用时延（Service Invoke Delay）算法，前者对传入的请求根据每台服务器当前所打开的连接数来分配；而在后者中，服务消费者缓存并计算所有服务提供者的服务调用时延，根据服务调用和平均时延的差值动态调整权重。

源 IP 哈希（Source IP Hash）算法实现请求 IP 粘滞连接，尽可能让消费者总是向同一提供者发起调用服务。这是一种有状态机制，也可以归为动态负载均衡算法。

## 4.4 服务路由

在集群化环境中，当客户端请求到达集群，如何确定由某一台服务器进行请求响应就是服务路由（Routing）问题。从这个角度讲，负载均衡也是一种路由方案，但是负载均衡的出发点是提供服务分发而不是解决路由问题，常见的静态、动态负载均衡算法也无法实现精细化的路由管理。服务路由的管理可以归为几个大类，包括直接路由、间接路由和路由规则。

### 4.4.1 直接路由

所谓直接路由就是服务的消费者需要感知服务提供者地址信息。服务消费者获取服务提供者地址的基本思路是通过配置中心或者数据库，当服务的消费者需要调用某个服务时，基于配置中心或者数据库中存储的目标服务的具体地址构建链路完成调用。这是常见的路由方案，但并不是一种好的方案。一方面，服务的消费者直接依赖服务提供者的具体地址，一旦在运行时服务提供者地址发生改变时无法在第一时间通知消费者，可能会导致服务消费者的相应变动，从而增强服务提供者和消费者之间的耦合度；另一方面，创建和维护配置中心或数据库持久化操作同样需要成本。



#### 4.4.2 间接路由

间接路由体现了解耦思想并充分发挥了发布-订阅（Publish-Subscribe）模式的作用，发布-订阅机制参考图 4-18。事件（Event）是整个结构能够运行所依赖的基本数据模型，围绕事件存在两个角色，即发布者和订阅者。发布者发布事件，订阅者关注自身所关注的事件，发布者和订阅者并不需要感知对方的存在，两者之间通过传输事件的基础设施进行完全解耦。

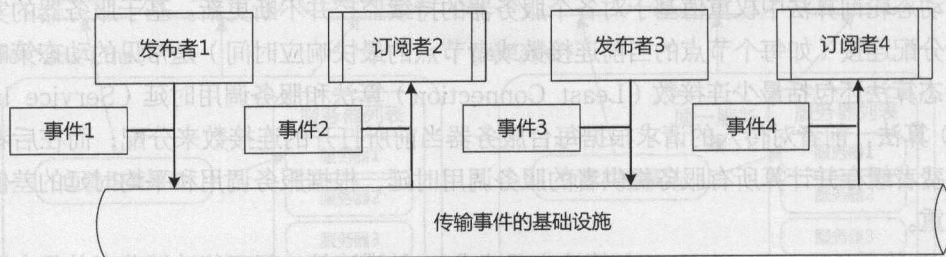


图 4-18 发布-订阅模式

在微服务架构中，实现间接路由的组件一般称为服务注册中心（Service Registration Center），服务注册中心从概念上讲就是发布-订阅模式中传输事件的基础设施，可以把服务的地址信息理解为事件的具体表现。

通过服务注册中心，服务提供者发布服务到注册中心，服务消费者订阅感兴趣的服务。服务消费者只需知道有哪些服务，而不需要知道服务具体在什么位置，从而实现间接路由。当服务提供者地址发生变化时，注册中心推送服务变化到服务消费者确保服务消费者使用最新的地址路由信息。同时，为了提高路由的效率和容错性，服务消费者可以配备缓存机制以加速服务路由，更重要的是当服务注册中心不可用时，服务消费者可以利用本地缓存路由实现对现有服务的可靠调用。

服务注册中心是间接路由的核心组件，在微服务架构中决定着服务发布与使用的方式，我们将在下一章中做单独介绍。

#### 4.4.3 路由规则

间接路由解决了路由解耦问题，面向全路由场景。在服务故障、高峰期导流、业务相关定制化路由等特定场景下，依靠间接路由提供的静态路由信息并不能满足需求，这就需要通过实现动态路由，动态路由可以通过路由规则（Routing Rule）进行实现。

路由规则常见的实现方案是白名单或黑名单，即把需要路由的服务地址信息（如服务 IP）放入可以控制是否可见的路由池中。更为复杂的场景可以使用 Python 等脚本语言实现各



种定制化条件脚本(Condition Script),如针对某些请求IP或请求服务URL中的特定语义进行过滤,也可以细化到运行时具体业务参数控制路由效率。

图4-19对服务路由相关策略做了总结,我们可以看到负载均衡和直接路由、间接路由、路由规则一样都可以看作是一种路由方案,路由方案为服务的消费者提供服务目标地址,并通过网络完成远程调用。

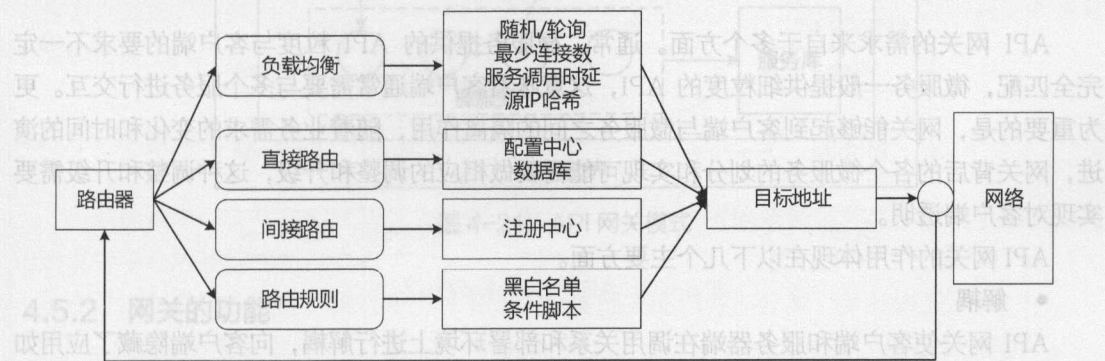


图 4-19 服务路由策略总览图

## 4.5 API 网关

在设计模式存在一种外观模式(Façade Pattern,也叫门面模式),其基本结构如图4-20所示。外观模式的设计意图在于为子系统的一组接口提供一个一致的入口,这个入口使得这一子系统更加容易使用。实现上用户界面不与系统耦合,而外观类与系统耦合。在层次化结构中,可以使用外观模式定义系统中每一层的入口。

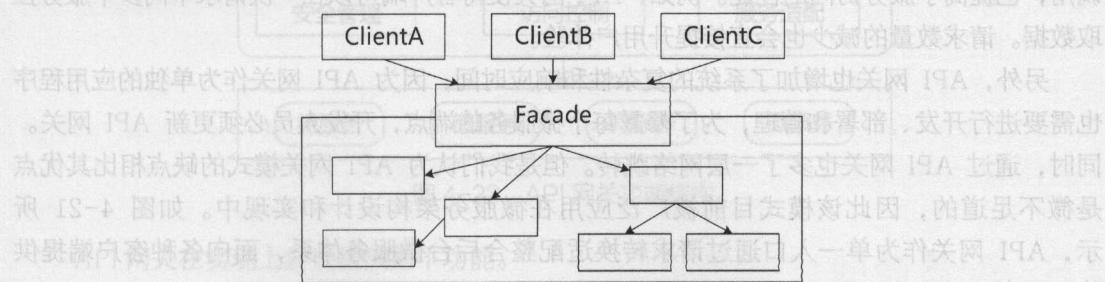


图 4-20 外观模式

API 网关本质上就是一种外观模式的具体实现,它是一种服务器端应用程序并作为系统访问的唯一入口。API 网关封装了系统内部架构,为每个客户端提供一个定制的 API。同时,它



可能还具有身份验证、监控、缓存、请求管理、静态响应处理等功能。在微服务架构中，API 网关的核心要点是，所有的客户端和消费端都通过统一的网关接入微服务，在网关层处理所有的非业务功能。

#### 4.5.1 网关的作用

API 网关的需求来自于多个方面。通常，微服务提供的 API 粒度与客户端的要求不一定完全匹配，微服务一般提供细粒度的 API，这意味着客户端通常需要与多个服务进行交互。更为重要的是，网关能够起到客户端与微服务之间的隔离作用，随着业务需求的变化和时间的演进，网关背后的各个微服务的划分和实现可能需要做相应的调整和升级，这种调整和升级需要实现对客户端透明。

API 网关的作用体现在以下几个方面。

- 解耦

API 网关使客户端和服务端在调用关系和部署环境上进行解耦，向客户端隐藏了应用如何被划分到微服务的细节。尽管微服务架构支持客户端直接与微服务交互的方式，但当需要交互的微服务数量较多时，解耦就成为一项核心需求。

- API 优化

API 网关向每个客户端提供最优的 API。在多客户端场景中，对于同一个业务请求，不同的客户端一般需要不同的数据。例如，对于同一个用户查询功能，PC 端会一次性获取所有数据，而对于手机 APP 而言，可以通过分页的方式逐步加载用户信息。

- 简化调用过程

由于能够对返回数据进行灵活处理，API 网关减少了请求往返次数，从而简化了客户端的调用，也提高了服务访问的性能。例如，API 网关使得客户端可以在一次请求中向多个服务拉取数据。请求数量的减少也会直接提升用户体验。

另外，API 网关也增加了系统的复杂性和响应时间，因为 API 网关作为单独的应用程序也需要进行开发、部署和管理，为了暴露每个微服务的端点，开发人员必须更新 API 网关。同时，通过 API 网关也多了一层网络跳转。但是我们认为 API 网关模式的缺点相比其优点是微不足道的，因此该模式目前被广泛应用在微服务架构设计和实现中。如图 4-21 所示，API 网关作为单一入口通过请求转换适配整合后台微服务体系，面向各种客户端提供统一服务。



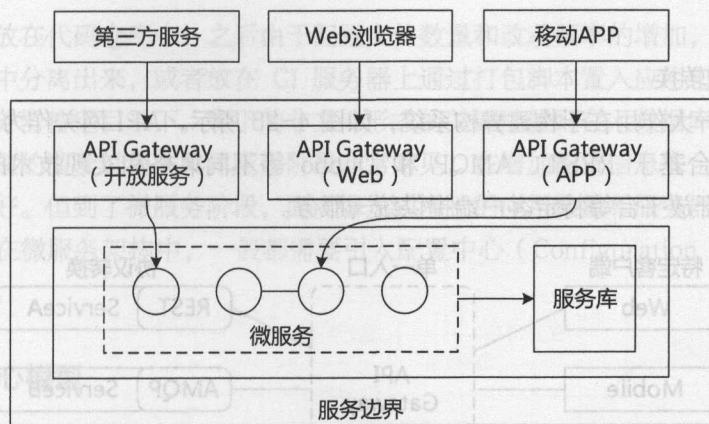


图 4-21 API 网关模式

## 4.5.2 网关的功能

API 网关的结构如图 4-22 所示。在这个结构背后，我们需要挖掘其所应具备的核心功能，从而发挥上文中所提到的各项作用。

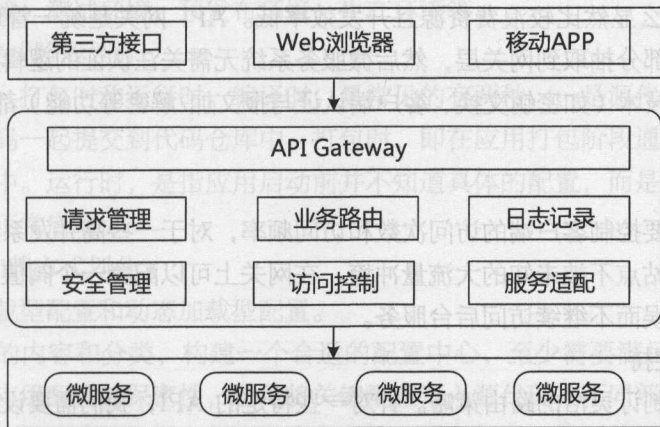


图 4-22 API 网关功能结构

API 网关在实现上需要提供如下功能。

- NIO 接入和异步接出

API 网关作为所有的客户端与微服务之间的桥梁，在两者之间多加了一层网络跳转。为了尽可能地消除这层网络跳转所带来的性能影响，API 网关在实现上需要提供 NIO 接入和异步接出的功能。参考 4.1 节中讨论的服务通信机制，我们明确这样的实现方式能够提供较高的通



信性能。

- 报文格式转换

API 网关的一大作用在于构建异构系统,如图 4-23 所示,API 网关作为单一入口,通过协议转换整合后台基于 REST、AMQP 和 Dubbo 等不同风格和实现技术的微服务,面向 Web、Mobile、开放平台等特定客户端提供统一服务。

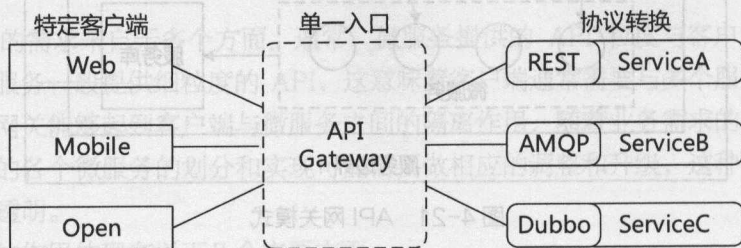


图 4-23 API 网关与异构系统

- 安全性控制

一般而言,无论对内网还是外网的接口都需要做用户身份认证,而用户认证在一些规模较大的系统都会采用统一的单点登录(Single Sign On, SSO)功能,如果每个微服务都要对接单点登录系统,那么显然比较浪费资源且开发效率低。API 网关是统一管理安全性的绝佳场所,可以将认证的部分抽取到网关层,然后微服务系统无需关注认证的逻辑只关注自身业务即可。常见的安全性技术(如密钥交换、客户端认证与报文加/解密等功能)都可以在 API 网关中加以实现。

- 访问控制

某些场景下需要控制客户端的访问次数和访问频率,对于一些高并发系统有时还会有限流的需求。为了防止站点不被未知的大流量冲垮,在网关上可以配置一个阈值,当请求数超过阈值时就直接返回错误而不继续访问后台服务。

- 业务路由支持

可以在网关层制订灵活的路由策略。针对一些特定的 API,我们需要设置白名单、路由规则等各类限制。而这些非业务功能的配置以及变更都可以在网关层单独操作。

## 4.6 配置管理

配置管理的需求在任何类型的系统中都存在,而且随着业务复杂度的上升和技术架构的演变,系统对配置方式也会提出越来越高的要求。例如,在单块系统中,配置管理方式典型的演变过程往往是这样的:刚开始是配置文件比较少,更新频率也不会太高,所以倾向于把所有配



置和源代码一起放在代码仓库中；之后由于配置文件数量和改动频率的增加，就会考虑将配置文件从代码仓库中分离出来，或者放在 CI 服务器上通过打包脚本置入应用包中，或者直接放到运行应用的服务器特定目录下，剩下的非文件形式的关键配置则存入数据库中。

上述配置管理的演变过程在单体应用阶段非常常见，在增加配置信息安全性的同时，也往往可以运行得很好。但到了微服务阶段，面对爆发式增长的应用数量和服务器数量，就显得无能为力。为此，在微服务架构中，一般都需要引入配置中心（Configuration Center）的设计思想和相关工具。

## 4.6.1 配置中心模型

所谓配置中心，简单来说就是一种统一管理各种应用配置的基础服务组件。本小结我们将给出配置中心的基本模型。

在讨论配置中心之前，我们先来梳理一下配置相关的内容和分类，参考如下。

- 按配置的来源划分

主要有源代码文件、数据库和远程调用。

- 按配置的适用环境划分

可分为开发环境、测试环境、预发布环境、生产环境等。

- 按配置的集成阶段划分

可分为编译时、打包时和运行时。编译时，最常见的有两种，一是源代码级的配置，二是把配置文件和源代码一起提交到代码仓库中。打包时，即在应用打包阶段通过某种方式将配置打入最终的应用包中。运行时，是指应用启动前并不知道具体的配置，而是先从本地或者远程获取配置，然后再正常启动。

- 按配置的加载方式划分

可分为单次加载型配置和动态加载型配置。

基于配置相关的内容和分类，构建一个合适的配置中心，至少需要满足如下 4 个核心需求：非开发环境下应用配置的保密性，避免将关键配置写入源代码；不同部署环境下应用配置的隔离性，比如非生产环境的配置不能用于生产环境；同一部署环境下的服务器应用配置的一致性，即所有服务器使用同一份配置；分布式环境下应用配置的可管理性，即提供远程管理配置的能力。

采用配置中心也就意味着采用集中式配置管理的设计思想（见图 4-24）。在集中式配置中心中，开发、测试和生产等不同的环境配置信息统一保存在配置中心中，这是一个维度。而另一个维度就是分布式集群环境，需要确保集群中同一类服务的所有服务器保存同一份配置文件并且能够同步更新。



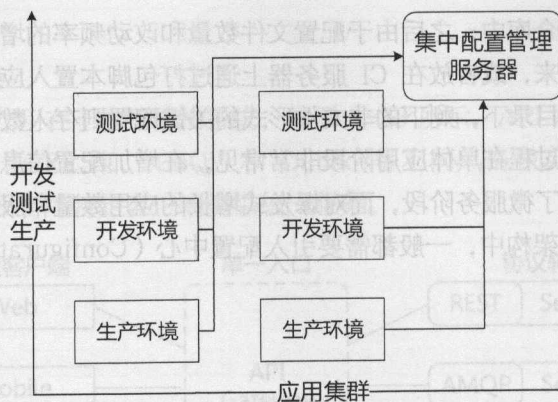


图 4-24 集中式配置中心示例图

## 4.6.2 分布式协调机制

配置中心在实现上需要满足如下几个要求：高效获取、实时感知、分布式访问。为了满足以上要求，配置中心通常需要依赖分布式协调（Distributed Coordination）机制，即通过一定的方法确保配置信息在分布式环境中的各个服务中能得到实时、一致的管理。本小节我们将基于目前业界主流的分布式协调框架 Zookeeper（<https://zookeeper.apache.org/>）介绍配置中心的实现思路。对 Zookeeper 的详细介绍不是本书的重点，读者可以参考相关资料<sup>[13]</sup>。

Zookeeper 的核心是一个精简的文件系统抽象，提供基于目录节点树方式的数据存储，以及一些额外的抽象操作，如排序、通知和监控等。如图 4-25 所示，Zookeeper 物理结构就是一个文件系统，每一个节点被称为 ZNode，代表该节点位于文件系统的路径，同时也用于存储数据。

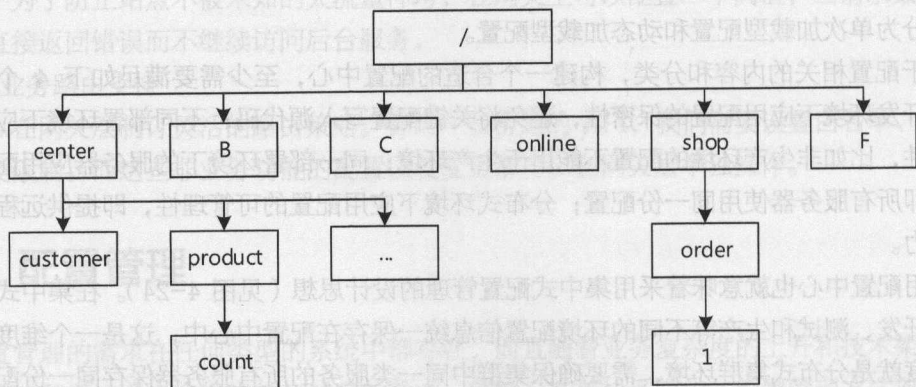


图 4-25 Zookeeper 物理结构



基于图 4-25 中的物理结构, Zookeeper 提供了一些核心组件用于实现分布式协调。例如, Zookeeper 对每个节点都可以设置监听器 (Watcher), 节点监听机制可以用来实现实时感知, 即当某一个节点的信息有任何变动时, 所有监听该节点的其他节点都可以实时获取通知, 从而做出响应。对于配置中心而言, 所有服务就是 Zookeeper 的客户端, 这些服务通过对包含配置信息的 Zookeeper 节点进行监听就能获取配置信息更新内容。基于 Zookeeper 实现配置中心的示意图如图 4-26 所示, 可以看到该机制同时也提供了分布式访问的效果。

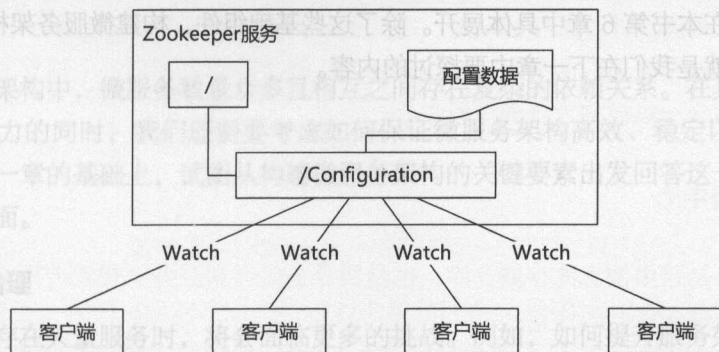


图 4-26 Zookeeper 实现配置中心示意图

对于配置信息的操作不外乎 CRUD, 在 Zookeeper 中, 修改配置可以拆分成删除和新增两个阶段, 所以整个配置信息的管理可以简化成新增、查询和删除操作。图 4-27 中包含了这三种操作的实现流程, 从中我们不难看出所有操作本质上就是对 Zookeeper 节点和监听器的合理利用。

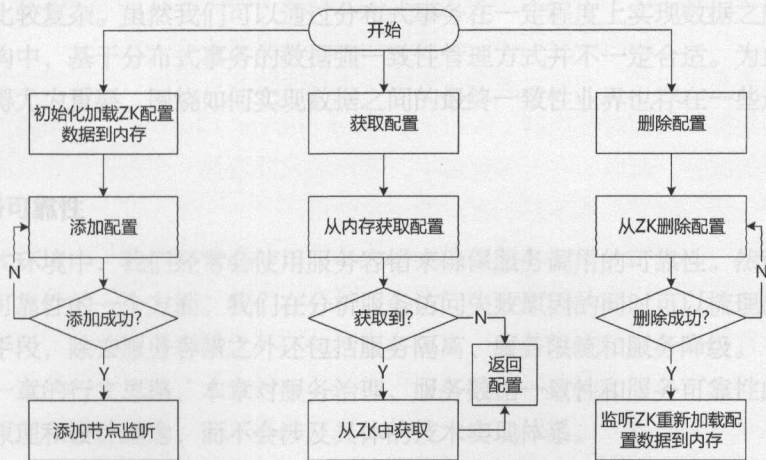


图 4-27 Zookeeper 实现配置信息 CRUD 流程图



## 4.7 本章小结

本章侧重介绍微服务架构实现过程中所需要具备的基础组件，包括服务通信、事件驱动、负载均衡、服务路由、API 网关和配置管理。这些组件一部分属于分布式系统通用的组件，而诸如 API 网关、配置管理等则是微服务架构区别一般分布式系统的特有组件。

本章对于微服务架构基础组件的描述重点在于介绍相关的需求和设计思想，具体的实现方案和工具我们将在本书第 6 章中具体展开。除了这些基础组件，构建微服务架构还需要考虑一些关键要素，这就是我们在下一章中要探讨的内容。

### 4.6.2 分布式协调机制

配置中心在实现上需要满足如下三个要求：高可用性、实时感知、强一致性。为了满足以上要求，配置中心通常需要具备分布式协调（Distributed Coordination）机制，即通过一定的方法确保配置信息在分布式环境中的各个服务中能得到实时、一致的管理。在本小节我们将基于 Zookeeper 实现配置中心，并介绍其分布式协调机制。Zookeeper 是一个分布式的、开源的、基于 Java 实现的分布式协调服务，它提供了丰富的接口，可以用于实现分布式系统中的各种协调操作，如命名空间管理、分布式锁、分布式队列、分布式选举等。Zookeeper 的物理结构由一个文件系统，每一个节点被称为 ZNode，代表该节点位于文件系统中的路径，同时也用于存储数据。

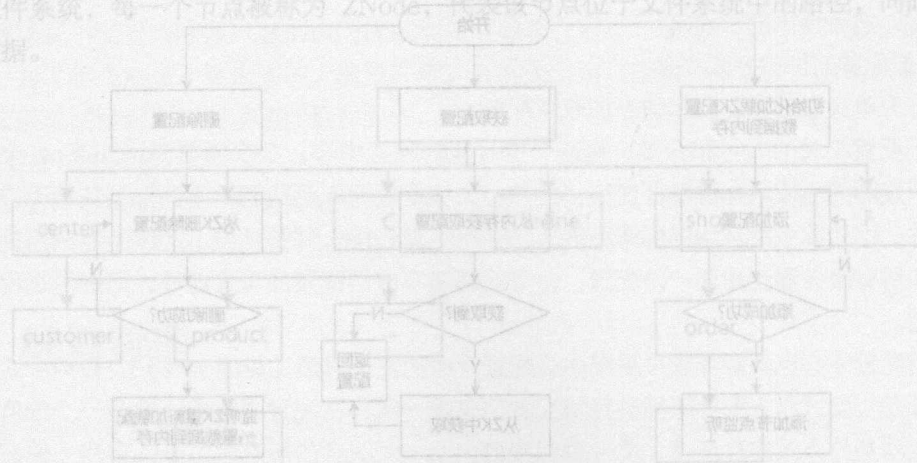


图 4-25 Zookeeper 物理结构图



# 微服务架构关键要素

在微服务架构中，微服务数量众多且相互之间存在复杂的依赖关系。在具备构建微服务架构基础组件能力的同时，我们还需要考虑如何保证微服务架构高效、稳定以及正确运行的问题。本章在上一章的基础上，试图从构建微服务架构的关键要素出发回答这一问题，具体表现在以下三个方面。

### 1. 服务治理

一个系统存在大量服务时，将会面临更多的挑战。例如，如何提升服务架构的可扩展性，如何进行服务监控和故障定位，如何实现对服务的有效划分和路由。服务治理（Service Governance）是应对这些挑战的统一方法；而在技术实现上，服务治理一般表现为服务发布与订阅机制以及实现该机制的服务注册中心。

### 2. 服务数据一致性

在单块系统中，方法调用之间的数据一致性可以通过事务来实现。但在分布式环境下，事情就会变得比较复杂。虽然我们可以通过分布式事务在一定程度上实现数据之间的一致性，但在微服务架构中，基于分布式事务的数据强一致性管理方式并不一定合适。为此，数据的最终一致性就显得尤为重要，围绕如何实现数据之间的最终一致性业界也存在一些通用的实现模式和方法。

### 3. 服务可靠性

在分布式环境中，我们经常会使用服务容错来确保服务调用的可靠性。然而，服务容错只是实现服务可靠性的一个方面。我们在分析服务访问失败原因的同时可以梳理出一系列实现服务可靠性的手段，除去服务容错之外还包括服务隔离、服务限流和服务降级。

如同上一章的行文思路，本章对服务治理、服务数据一致性和服务可靠性的讨论也主要围绕其基本的原理和设计理念，而不会涉及具体的技术实现体系。



## 5.1 服务治理

在微服务架构中，服务治理（Service Governance）可以说是最为关键的一个要素，因为各个微服务需要通过服务治理实现自动化的注册（Registration）和发现（Discovery）。

服务治理的需求来自于服务的数量。在服务数量并不是太多的场景下，服务消费者获取服务提供者地址的基本思路是通过配置中心，当服务的消费者需要调用某个服务时，基于配置中心中存储的目标服务的具体地址构建链路，完成调用。这是常见的路由方案，但并不是一种好的方案。为了实现微服务架构中的服务注册和发现，通常都需要构建一个独立的媒介来管理服务的实例，这个媒介一般被称为服务注册中心。

### 5.1.1 服务注册中心

服务注册中心是路由信息的存储仓库，也是服务提供者和服务消费者进行交互的媒介，充当着服务注册和发现服务器的作用。

#### 1. 注册中心核心功能

注册中心应该具备发布-订阅功能，体现在服务提供者可以根据服务的元数据发布服务，而服务消费者则通过对自己感兴趣的服务进行订阅并获取包括服务地址在内的各项元数据。发布-订阅的功能还体现在数据变更推送，即当注册中心服务定义发生变化时，主动推送变更到该服务的消费者从而实现间接路由。由于服务提供者和服务消费者同时依赖于注册中心，就需要确保数据一致性，在任何时候服务提供者和消费者都应该看到同一份数据。作为发布-订阅模式的一种实现，注册中心的这些功能都与该模式的基本原理保持一致。

为了确保服务高可用性，一般也需要注册中心本身保持高可用性，也就意味着注册中心本身需要构建对等集群。所谓对等集群（Peer-to-peer Server Cluster），是指集群中所有的服务器都提供同样的服务。所以，在对等集群中，客户端只要连接一个服务器完成服务注册和发现即可，任何一台服务器宕机都不影响客户端正常使用。

另一方面，作为注册中心的客户端程序，一般会嵌入在服务提供者和服务消费者的应用程序中。在应用程序运行时，服务提供者的注册中心客户端程序会向注册中心注册自身提供的服务，而服务消费者的注册中心客户端程序则从注册中心查询当前订阅的服务信息并周期性的刷新服务状态。同时，为了提高服务路由的效率和容错性，服务消费者可以配备缓存机制以加速服务路由，更重要的是当服务注册中心不可用时，服务消费者可以利用本地缓存路由实现对现有服务的可靠调用。图 5-1 展示了注册中心客户端与注册中心的交互过程。



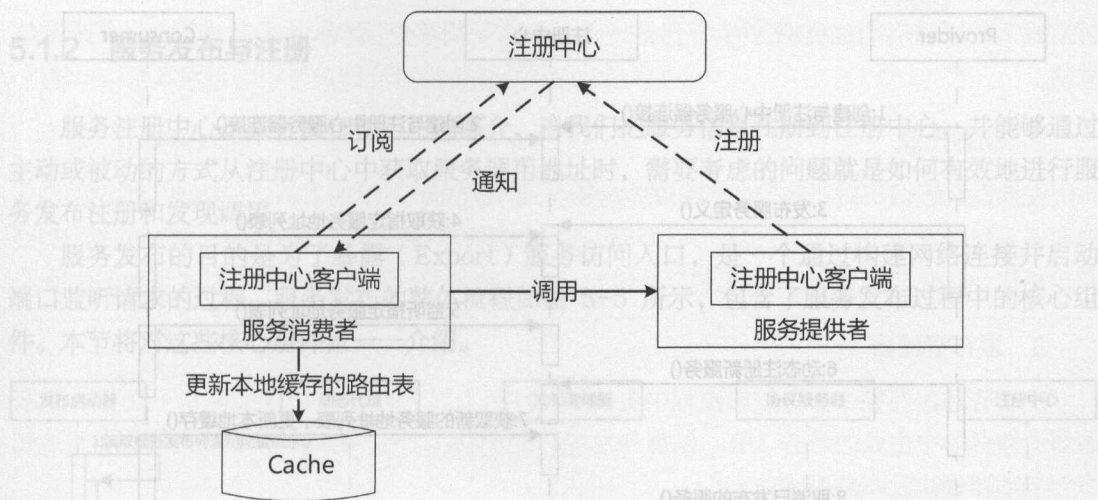


图 5-1 注册中心客户端与注册中心交互过程

基于以上分析,我们可以对注册中心进行抽象和建模(见图 5-2),这里的客户端既可以是服务提供者,也可以是服务消费者。接下去我们将讨论如何实现该模型。

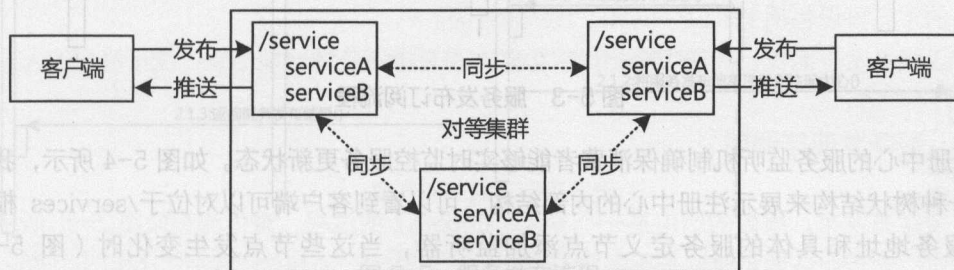


图 5-2 注册中心模型

## 2. 注册中心实现策略

基于注册中心各项特性所采取的注册中心基本实现流程如图 5-3 所示。该图中服务的提供者和消费者可以使用一定的通信机制与注册中心服务器建立连接并维持心跳检测,通过注册中心提供的操作接口分别完成发布和动态更新服务定义、获取指定服务地址列表、取消服务发布、获取服务地址更新等功能。





图 5-3 服务发布订阅流程

注册中心的服务监听机制确保消费者能够实时监控服务更新状态。如图 5-4 所示, 我们假定以一种树状结构来展示注册中心的内部结构, 可以看到客户端可以对位于 /services 根目录下的服务地址和具体的服务定义节点添加监听器, 当这些节点发生变化时 (图 5-4 中 Service2 节点表示取消注册, Service5 和 Service6 节点表示新增注册), 注册中心就能触发各个客户端监听器中的回调函数确保更新通知到每一个客户端。

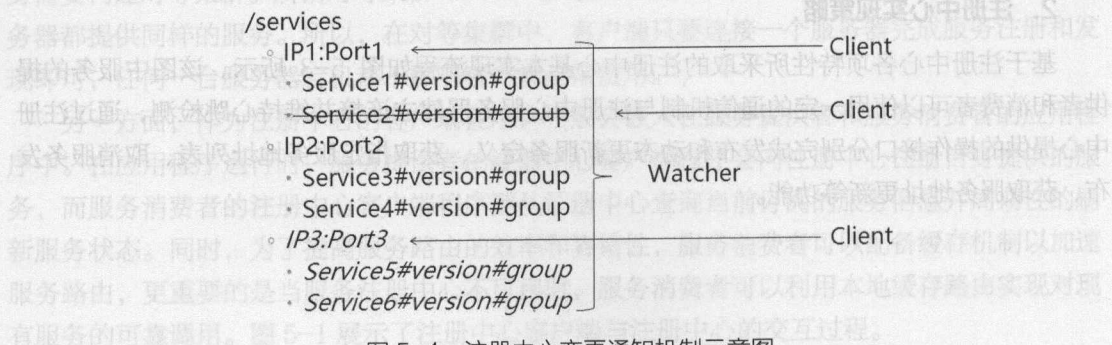


图 5-4 注册中心变更通知机制示意图



5.1.2 服务发布与注册

服务注册中心是服务发布和引用的媒介，当我们把服务信息注册到注册中心，并能够通过主动或被动的方式从注册中心中获取服务调用地址时，需要考虑的问题就是如何有效地进行服务发布注册和发现调用。

服务发布的目的是为了暴露（Export）服务访问入口，是一个通过构建网络连接并启动端口监听请求的过程。服务发布的整体流程如图 5-5 所示，包含了服务发布过程中的核心组件。本节将对这些核心组件做一一介绍。

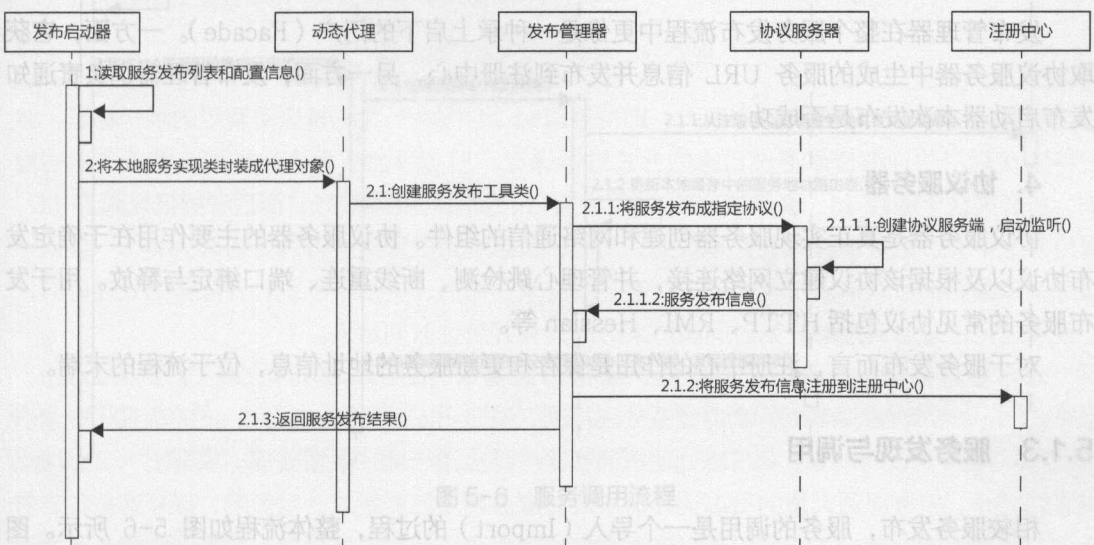


图 5-5 服务发布流程

1. 发布启动器

发布启动器（Launcher）的作用是确定服务发布形式并启动发布平台。服务的发布形式常见有三种，即配置化、API 调用和使用注解。通过以 XML 为代表的配置化工具，服务框架对业务代码零侵入，扩展和修改方便，同时配置信息修改能够实时生效；而通过 API 调用方式，服务框架对业务代码侵入性较强，修改代码之后需要重新编译才能生效；注解方式中，服务框架对业务代码零侵入，扩展和修改也比较方便，但修改配置需要重新编译代码。以上三种方式各有利弊，一般我们倾向于使用基于配置的方式，但在涉及系统之间集成时，由于需要使用服务框架中较底层的服务接口，API 调用可能是唯一的选择。

发布平台的启动与所选择的发布方式密切相关。在使用配置化发布方式时，通常会借助于诸如 Spring 的容器进行服务实例的配置和管理，容器的正常启动意味着发布平台的启动，注



解方式下的平台启动也类似。而对于 API 调用而言，简单使用 main 函数进行启动是通常的做法。

## 2. 动态代理

在涉及远程调用时，我们通常会在本地服务实现的基础上添加动态代理功能。通过动态代理实现对服务发布进行动态拦截，可以对服务发布行为本身进行封装和抽象，同时也便于扩展和定制化。JDK 自带的 Proxy 机制以及诸如 javassist 的字节码编辑库都可以实现动态代理。

## 3. 发布管理器

发布管理器在整个服务发布流程中更像是一种承上启下的门户（Facade）。一方面，它获取协议服务器中生成的服务 URL 信息并发布到注册中心，另一方面，发布管理器也负责通知发布启动器本次发布是否成功。

## 4. 协议服务器

协议服务器是真正实现服务器创建和网络通信的组件。协议服务器的主要作用在于确定发布协议以及根据该协议建立网络连接，并管理心跳检测、断线重连、端口绑定与释放。用于发布服务的常见协议包括 HTTP、RMI、Hessian 等。

对于服务发布而言，注册中心的作用是保存和更新服务的地址信息，位于流程的末端。

### 5.1.3 服务发现与调用

相较服务发布，服务的调用是一个导入（Import）的过程，整体流程如图 5-6 所示。图中我们可以看到服务调用流程与服务发布流程呈对称结构，所包含的组件包括以下几类。

#### 1. 调用启动器

调用启动器的作用就是确定服务的调用形式并启动调用平台，该组件使用的策略与发布启动器一样，不再重复介绍。

#### 2. 动态代理

动态代理完成本地接口到远程调用的转换，导入服务提供者接口 API 和服务信息并生成远程服务的本地动态代理对象，从而将本地 API 调用转换成远程服务调用并返回调用结果。

#### 3. 调用管理器

调用管理器具备缓存功能，保存着服务地址的缓存信息。当从注册中心获取服务提供者地



址信息时，调用管理器根据需要更新本地缓存，确保在注册中心不可用的情况下，调用启动器仍然可以从本地缓存中获取服务提供者的有效地址信息。

4. 协议客户端

协议客户端根据服务调用指定的协议类型创建客户端并发起连接请求，负责与协议服务器进行交互并获取调用结果。服务调用流程如图 5-6 所示。

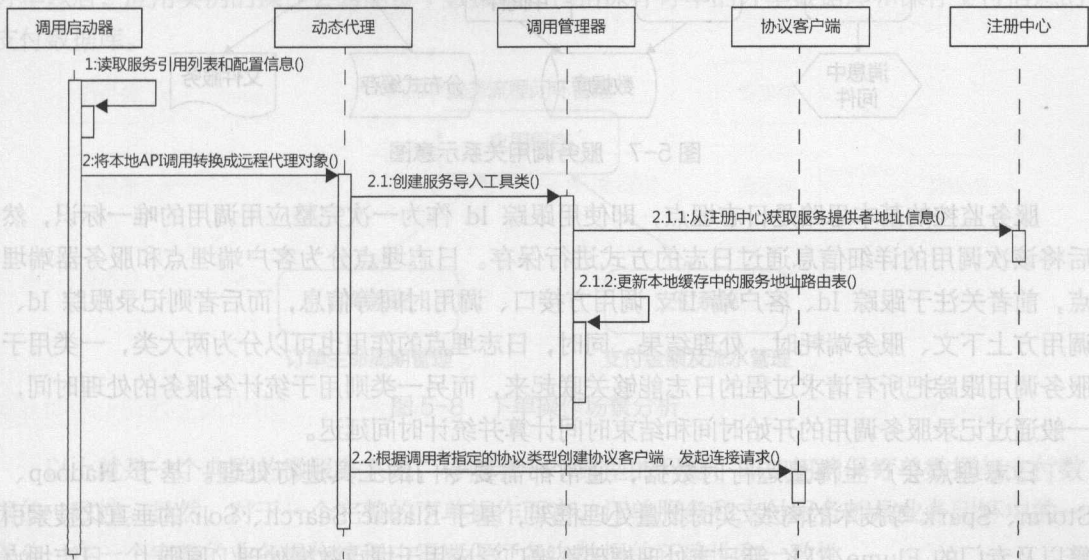


图 5-6 服务调用流程

在服务调用过程中，实现了从本地缓存获取服务路由、序列化请求消息并封装成协议消息、发送协议消息并同步等待或注册监听器回调、反序列化应答消息并唤醒业务线程或触发监听器等分布式服务的基本步骤。如果调用超时或失败，将采用集群容错机制。至此，整个服务发布和调用过程形成闭环。

5.1.4 服务监控

在分布式环境下，围绕某个业务链的所有服务之间的调用关系可能非常复杂，图 5-7 所示的是服务调用关系的一种表现。我们可以看到服务中间件、数据库、缓存、文件系统以及其他服务之间都可能存在依赖关系。为了确保系统运行时这些依赖关系的稳定性和可用性，服务调用路径、服务调用业务数据、服务性能数据都是需要监控的内容，以便进行系统故障的预防和定位。



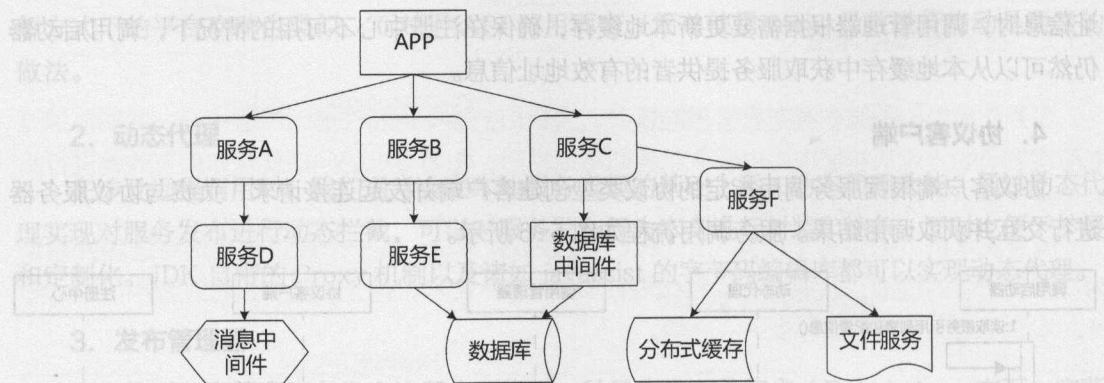


图 5-7 服务调用关系示意图

服务监控的基本思路是日志埋点，即使用跟踪 Id 作为一次完整应用调用的唯一标识，然后将该次调用的详细信息通过日志的方式进行保存。日志埋点分为客户端埋点和服务器端埋点，前者关注于跟踪 Id、客户端 IP、调用方接口、调用时间等信息，而后者则记录跟踪 Id、调用方上下文、服务端耗时、处理结果。同时，日志埋点的作用也可以分为两大类，一类用于服务调用跟踪把所有请求过程的日志能够关联起来，而另一类则用于统计各服务的处理时间，一般通过记录服务调用的开始时间和结束时间计算并统计时间延迟。

日志埋点会产生海量运行时数据，通常都需要专门的工具进行处理。基于 Hadoop、Storm、Spark 等技术的离线/实时批量处理框架，基于 Elastic Search、Solr 的垂直化搜索引擎以及专门的 Flume/ELK 等日志处理框架都被广泛应用于埋点数据处理。原则上，日志埋点越详细越好，但考虑到大量的日志读写请求会对服务调用性能造成一定影响，保存这些日志信息也需要较大的存储成本，所以在日志埋点过程中也可以使用抽样思想，并不一定将所有的场景都进行日志埋点。

随着业务规模扩大，针对如何管理分布式服务，容量规划、资源利用率、服务上下线管理等问题是开发和运维人员都面临的挑战。服务治理的目标在于保障线上服务运行质量，治理的对象是基于统一分布式服务框架开发的各项业务服务，服务治理在定位上关注服务运行时状态、细粒度治理，服务限流/降级、服务动态路由和灰度发布是服务治理的基本策略。具体实现上，可以采用通过注册中心对服务依赖进行分析，结合运行时调用关系，梳理不合理的依赖和调用路径，优化服务架构。实时收集服务调用日志，分析、汇总、存储和展示，方便开发和运维人员进行实时故障诊断，同时执行服务运行时治理方案，包括限流降级、路由、统一配置等在线调整。服务监控属于微服务管理体系的建设内容，关于服务监控的原理和工具我们在第 7 章中会有进一步介绍。



## 5.2 数据一致性

试想图 5-8 所示的业务场景，系统中存在两个服务，一个为订单服务，另一个为支付服务。图 5-8 所示的应用程序为了实现业务流程闭环管理，涉及两个主要的步骤，用户下单并将订单详细信息记录到订单数据库中，同时通过支付系统产生支付记录。也就是说当用户下了订单以后，应用实例的操作会跨越多个数据库，包括保存订单的订单数据库和保存支付信息的支付数据库。

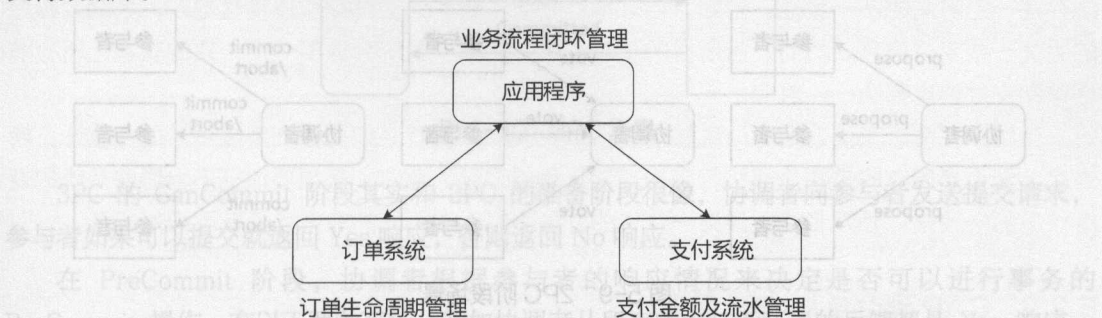


图 5-8 下单操作场景分析

以上就是一个典型的微服务系统，伴随着一个典型的问题，即如何确保订单数据与支付数据的一致性。显然，对于一个完整的下单操作而言，订单服务和支付服务都是业务闭环中的一部分，在一个完整的业务操作流程中需要保证各自数据的正确性和一致性。

### 5.2.1 分布式事务

对于数据一致性问题，我们首先想到的就是使用事务（Transaction）机制。事务包括本地事务，也包括复杂环境下的分布式事务。

#### 1. 本地事务与分布式事务

传统单机应用一般都会使用一个关系型数据库，好处是可以使用 ACID 事务特性。为保证数据一致性我们只需要开启一个事务，执行数据更新操作，然后提交事务或回滚事务。更进一步，借助于 Spring 等数据访问技术和框架，我们只需要关注引起数据改变的业务本身即可。

随着组织规模不断扩大，业务量不断增长，单块应用和数据库已经不足以支持庞大的业务量和数据量，这个时候需要对服务和数据进行拆分，就出现了图 5-8 所示的一个应用需要同时访问两个或两个以上的数据库的情况。显然，图 5-8 所示的场景并不能依靠本地事务得以解决。这时候可以借助于分布式事务来保证一致性，也就是通常所说的两阶段提交协议（Two



Phase Commit, 2PC) 和三阶段提交协议 (Three Phase Commit, 3PC)。

### (1) 两阶段提交协议

两阶段提交, 顾名思义分成两个阶段, 先由一方提议并收集其他节点的反馈(准备阶段), 再根据反馈决定提交或中止事务(执行阶段)。一般将提议的节点称为协调者 (Coordinator), 其他参与决议的节点称为参与者 (Participants)。图 5-9 展示了协调者发起一个提议分别询问各参与者是否接受的场景, 然后协调者根据参与者的反馈, 提交或中止事务。如果参与者全部同意则提交, 只要有一个参与者不同意就中止。

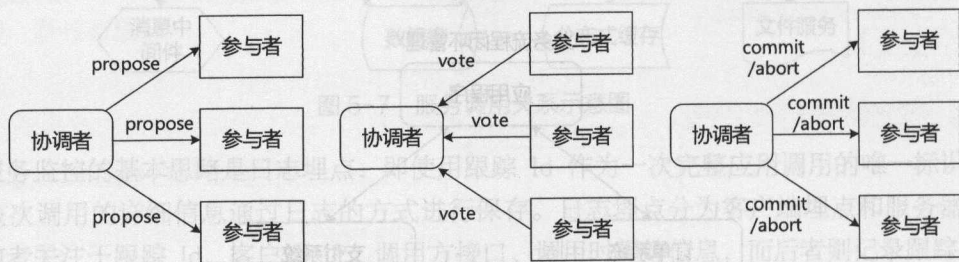


图 5-9 2PC 阶段场景

二阶段提交看起来确实能够提供原子性的操作, 但不幸的是, 二阶段提交存在一些明显的缺点。

- 同步阻塞问题

执行过程中, 所有参与节点都是事务阻塞型的。当参与者占有公共资源时, 其他第三方节点访问公共资源不得不处于阻塞状态。

- 单点故障

由于协调者的重要性, 一旦协调者发生故障, 参与者会一直阻塞下去。尤其在第二阶段, 协调者发生故障, 那么所有的参与者还都处于锁定事务资源的状态中, 而无法继续完成事务操作。

- 数据不一致

在二阶段提交中, 当协调者向参与者发送提交请求之后发生了局部网络异常, 或者在发送提交请求过程中协调者发生了故障, 就会导致只有一部分参与者接收到了提交请求。而这部分参与者接到提交请求之后就会执行提交操作, 但是其他未接到提交请求的机器则无法执行事务提交。于是, 整个分布式系统便出现了数据不一致性的现象。

由于二阶段提交存在以上缺陷, 业界在二阶段提交的基础上做了改进, 提出了三阶段提交。

### (2) 三阶段提交协议

三阶段提交协议是二阶段提交协议的改进版, 与两阶段提交相比, 三阶段提交有两个改动点。其一是同时在协调者和参与者中都引入超时机制, 其二则是把二阶段提交中的准备阶段再次一分为二, 这样三阶段提交就有 CanCommit、PreCommit、DoCommit 三个阶段。三阶段



提交中场景分析如图 5-10 所示。

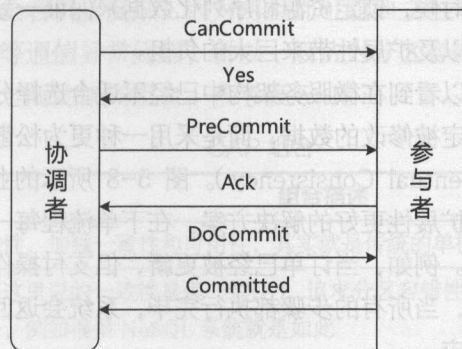


图 5-10 3PC 场景

3PC 的 CanCommit 阶段其实和 2PC 的准备阶段很像，协调者向参与者发送提交请求，参与者如果可以提交就返回 Yes 响应，否则返回 No 响应。

在 PreCommit 阶段，协调者根据参与者的响应情况来决定是否可以进行事务的 PreCommit 操作。有以下两种可能：假如协调者从所有的参与者获得的反馈都是 Yes 响应，那么就会执行事务的预执行；假如有任何一个参与者向协调者发送了 No 响应，或者等待超时之后协调者都没有接到参与者的响应，那么就执行事务的中断。

DoCommit 阶段进行真正的事务提交，也可以分为以下两种情况：执行提交和中断事务。当协调者没有接收到参与者发送的 ACK 响应（可能是接受者发送的不是 ACK 响应，也可能响应超时），那么就会执行中断事务。

相对于 2PC，3PC 主要解决的是单点故障问题，并减少阻塞。因为一旦参与者无法及时收到来自协调者的信息就会默认执行提交，而不会一直持有事务资源并处于阻塞状态。

## 2. 传统分布式事务的问题

分布式事务能解决一部分数据一致性的问题，但在微服务架构中，传统分布式事务并不是实现数据一致性的最佳选择。

首先，对于微服务架构来说，数据访问变得更加复杂，这是因为数据都是微服务私有的，唯一可访问的方式就是通过 API。这种打包数据访问的方式使得微服务之间松耦合，并且彼此之间高度独立，从而非常容易进行性能扩展。

其次，不同的微服务经常使用不同的数据库。序列化以及锁的使用，都是当且仅当多个服务使用相同数据库的前提下才能正常工作。在微服务架构中，服务会产生各种不同类型的数据，关系型数据库并不一定是最佳选择，很多微服务都会采用 SQL 和 NoSQL 结合的模式，如搜索引擎、图数据库等 NoSQL 数据库大多数并不支持 2PC 和 3PC。



而且, 服务在使用锁的过程中, 持有锁的时间都是非常短的。但当数据被拆分了或者在不同的数据库存在重复数据的时候, 锁定资源和序列化数据来保证一致性就会变成一个非常昂贵的操作, 会给系统的吞吐量以及扩展性带来巨大的负担。

通过以上分析, 我们可以看到在微服务架构中已经不适合选择分布式事务。因此, 当下主流的分布式应用大都不会锁定被修改的数据, 而是采用一种更为松散的方式来维护一致性, 也就是所谓的最终一致性 (Eventual Consistency)。图 5-8 所示的业务场景, 将下单流程采用最终一致性模型实现是一种扩展性更好的解决方案。在下单流程每一步执行的过程中, 整个系统存在一定时间的不一致性。例如, 当订单已经被更新, 但支付操作还没落地时, 系统会暂时的丢失一些支付信息。然而, 当所有的步骤都执行完毕, 系统会返回一个一致的状态, 所有的订单和支付信息都会一一对应。

当然, 在概念上说明最终一致性模型很简单, 但开发者必须要保证系统最后的一致性。换言之, 无论是所有的步骤全部执行完毕或者是有的步骤失败时, 必须要保证不会影响系统的最终状态的一致性。开发者实现最终一致性的方案可以根据不同的业务场景做不同的选择。

## 5.2.2 CAP 理论与 BASE 思想

在讨论如何实现最终一致性之前, 先介绍分布式系统中经典的 CAP 理论和 BASE 思想。

### 1. CAP 理论

CAP 理论<sup>[6]</sup>指的是在一个分布式系统中, 无法同时实现一致性 (Consistency, C)、可用性 (Availability, A) 和分区容错性 (Partition Tolerance, P)。

图 5-11 所示的分布式环境中, 一致性 (C) 是指在分布式系统中的所有数据备份在同一时刻是否拥有同样的值; 可用性 (A) 是指在集群中一部分节点故障后, 集群整体是否还能正常响应请求; 分区容错性 (P) 中的分区相当于对通信的时限要求, 系统如果不能在一定时限内达成数据一致性, 就意味着发生了分区的情况, 也就是说整个分布式系统不再互联。

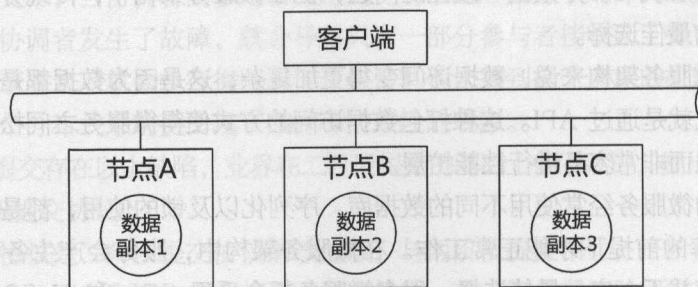


图 5-11 典型的分布式系统



现实环境中，出现分区（P）就相当于分布式环境中的节点分成了两组，彼此之间无法通信。这种情况比其中一组全宕机更复杂，因为两组都可能独立对外提供服务。由于当前的网络硬件肯定会出现延迟丢包等通信异常问题，所以一般认为分区容错性必须实现。关于一致性、可用性和分区容错性三种之间的排列组合，参考表 5-1。

表 5-1

CAP 组合

组合维度	组合描述
CA	放弃分区容错性，加强一致性和可用性，其实就是传统的单机数据库的选择
AP	放弃一致性（这里说的一致性 <strong>是强一致性</strong> ），追求分区容错性和可用性，这是很多分布式系统设计时的选择，例如很多 NoSQL 系统就是如此
CP	放弃可用性，追求一致性和分区容错性，基本不会选择，网络问题会直接让整个系统不可用

## 2. BASE 思想

eBay 的架构师丹·普里斯特（Dan Pritchett）通过对大规模分布式系统的实践总结，在 ACM 上发表文章提出 BASE 理论<sup>[17]</sup>。BASE 理论是对 CAP 理论的延伸，核心思想是即使无法做到强一致性，应用可以采用适合的方式达到最终一致性。BASE 是指基本可用（Basically Available）、软状态（Soft State）和最终一致性（Eventual Consistency）。

基本可用是指分布式系统在出现故障的时候，允许损失部分可用性，即保证核心可用。例如，我们在本章后续内容中会介绍的服务限流和服务降级都是基本可用思想的具体体现。

软状态是指允许系统存在中间状态，而该中间状态不会影响系统整体可用性。分布式存储中一般一份数据都会有若干个副本，允许不同节点间副本同步的延时就是软状态的体现。关系型数据库中如 Mysql 的复制功能也是该思想的一种体现。

最终一致性是指系统中的所有数据副本经过一定时间后，最终能够达到一致的状态。CAP 的一致性就是强一致性，这种一致性级别是最符合用户直觉的，它要求系统写入什么，读出来的也会是什么，用户体验好，但实现起来往往对系统的性能影响较大。弱一致性和强一致性相反，弱一致性级别约束了系统在写入成功后，不承诺立即可以读到写入的值，也不会承诺多久之后数据能够达到一致，但会尽可能地保证到某个时间级别（比如秒级别）后，数据能够达到一致状态。BASE 中的最终一致性可以看作是弱一致性的一种特殊情况。

CAP 理论与 BASE 思想实际上是有关联的，图 5-12 所示为 CAP 理论的另一种视角，我们看到一致性、可用性和分区容错性之前存在一定的交集。其中一致性与可用性的交集就是以 ACID 为主要特征的传统关系型数据库，可用性与分区容错性相交就体现了 BASE 思想，而一致性与分区容错性之间我们认为没有关联的，即不可能在这两者之间找到合适的交点。图 5-12 体现的 CAP 理论与 BASE 思想之间的交集与表 5-1 中的描述高度一致。



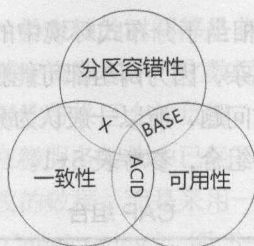


图 5-12 CAP 理论与 BASE 思想的关系

以上关于 CAP 理论与 BASE 思想讨论的意义在于即使无法做到强一致性，我们也可以采用合适的方式达到最终一致性，这点在微服务架构中表现得尤为明显。接下去我们就来介绍实现最终一致性的一些具体方法，包括可靠事件模式、Sagas 长事务模式、补偿模式、TCC 模式、最大努力通知模式和人工干预模式。

### 5.2.3 可靠事件模式

在本节中，我们将结合图 5-8 所示的业务场景，分析基于可靠事件模式的设计思路以及实现方案。

#### 1. 基本思路

在可靠事件模式中，当我们尝试将订单和支付两个微服务进行分别管理的时候，需要找到一种媒介用于在这两个服务之间进行数据传递。一般而言，消息中间件（Message-Oriented Middleware, MOM）适合扮演数据传递媒介的角色。引入消息中间件之后的下单操作流程可以拆分成如下三个步骤。

##### (1) 用户下单

当用户使用订单服务下单时，一方面订单服务需要对所产生的订单数据进行持久化操作，另一方面，它也需要同时发送一条创建订单的消息到消息中间件。图 5-13 展示了这一过程。

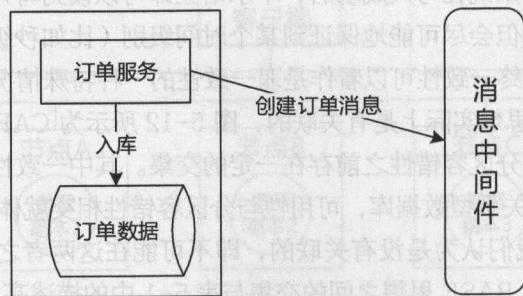


图 5-13 订单服务发送创建订单消息到消息中间件



(2) 交易支付

当消息中间件接收到订单创建消息，就会把该消息发送到支付服务。支付服务接收到订单创建消息之后，同样对该消息进行业务处理并持久化。如图 5-14 所示，当所有关于支付相关的业务逻辑执行完成之后，支付服务需要向消息中间件发送一条支付成功消息。

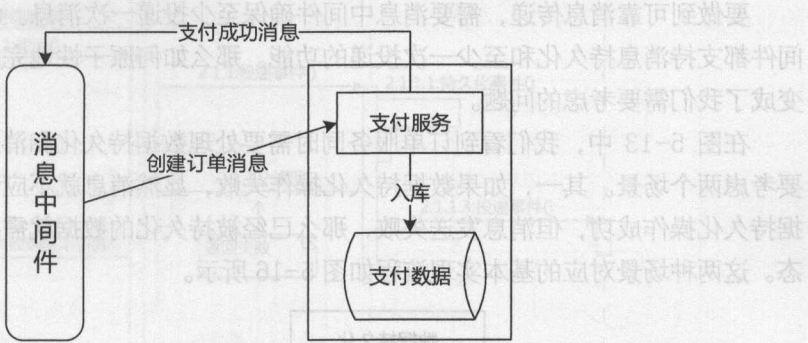


图 5-14 支付服务处理订单消息并返回处理结果

(3) 订单更新

支付成功消息通过消息中间件传递到订单服务时，订单服务根据支付的结果处理后续业务流程，一般会涉及订单状态的更新、向用户发送通知等内容（见图 5-15）。

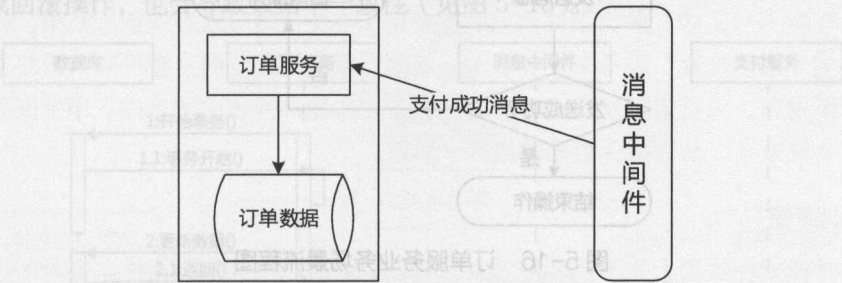


图 5-15 订单服务消费支付成功消息

2. 关键点

图 5-13、图 5-14 和图 5-15 展示了基于消息传递所形成的订单服务与支付服务之间的闭环管理。在正常情况下，该流程能够满足业务需求。但我们仔细分析整个过程，不难发现可能存在如下问题。

- 某个服务在更新了业务实体后发布消息失败；
- 虽然服务发布事件成功，但是消息中间件未能正确推送事件到订阅的服务；
- 接受事件的服务重复消费事件。



要做到可靠消息传递，需要消息中间件确保至少投递一次消息。幸好，目前主流的消息中间件都支持消息持久化和至少一次投递的功能。那么如何原子性地完成业务操作和发布消息就变成了我们需要考虑的问题。



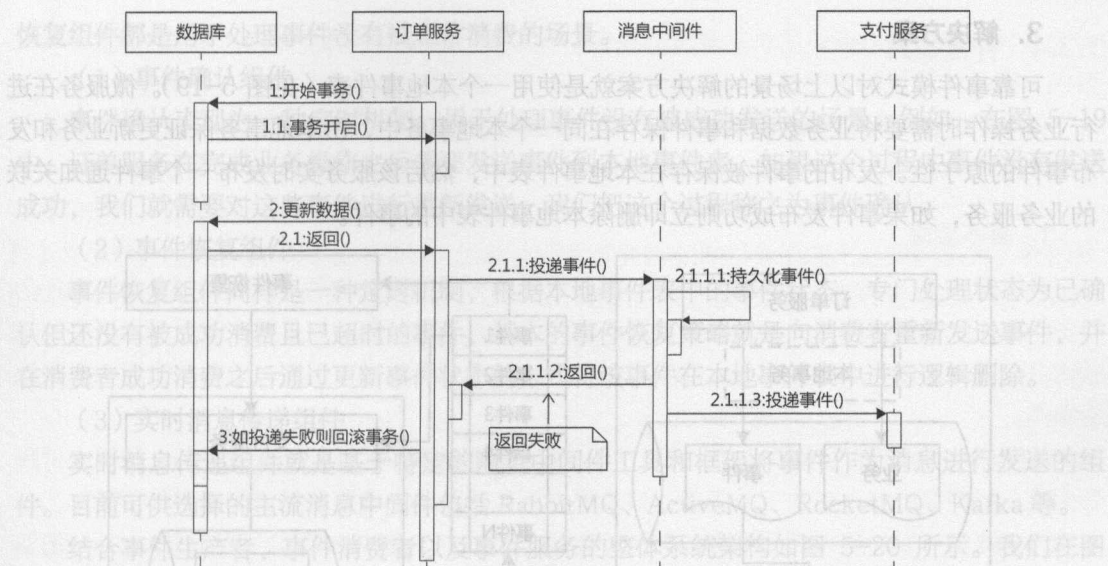


图 5-17 网络通信引起数据不一致

图 5-16 所示的流程还存在另一种场景问题，即当订单服务投递消息之后等待消息返回，但当消息真正返回时，订单服务自身发生错误导致其不可用。显然，这种场景下订单服务无法执行提交或回滚操作，也会导致数据不一致性（见图 5-18）。

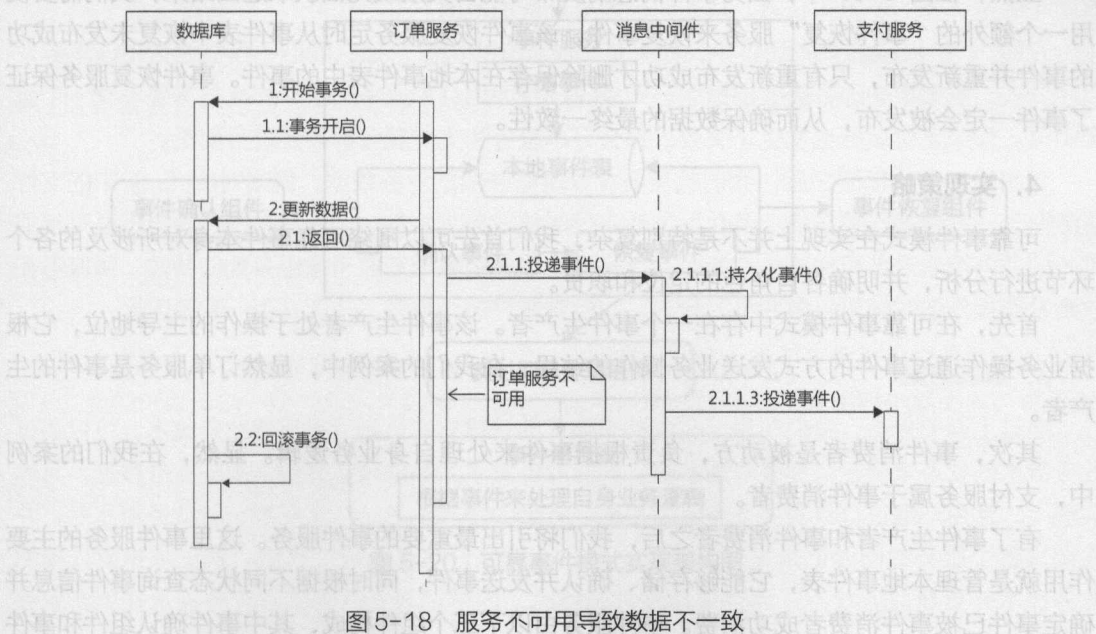


图 5-18 服务不可用导致数据不一致



### 3. 解决方案

可靠事件模式对以上场景的解决方案就是使用一个本地事件表（见图 5-19）。微服务在进行业务操作时需要将业务数据和事件保存在同一个本地事务中，由本地事务保证更新业务和发布事件的原子性。发布的事件被保存在本地事件表中，然后该服务实时发布一个事件通知关联的业务服务，如果事件发布成功则立即删除本地事件表中的事件。

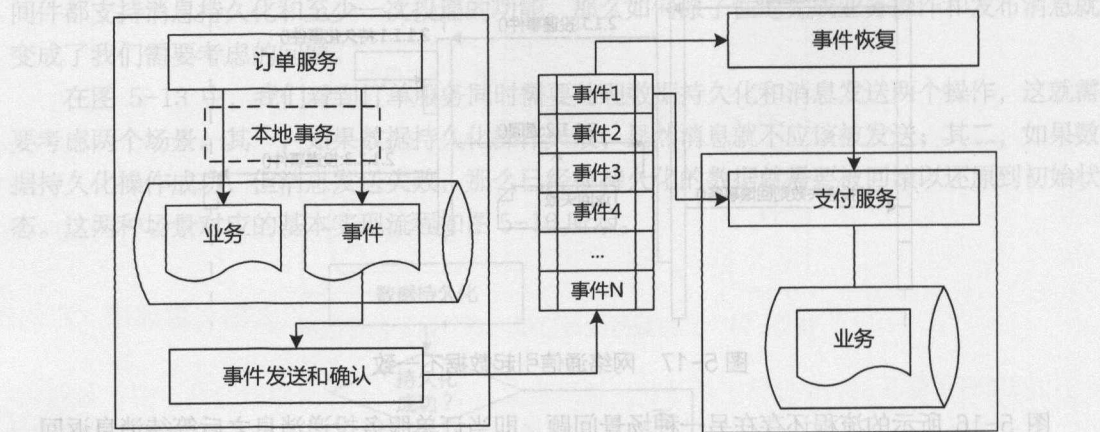


图 5-19 本地事件表

显然，在图 5-19 中，因为事件消息的发布可能会失败或无法获取返回结果，我们需要使用一个额外的“事件恢复”服务来恢复事件，该事件恢复服务定时从事件表中恢复未发布成功的事件并重新发布，只有重新发布成功才删除保存在本地事件表中的事件。事件恢复服务保证了事件一定会被发布，从而确保数据的最终一致性。

### 4. 实现策略

可靠事件模式在实现上并不是特别复杂，我们首先可以围绕可靠事件本身对所涉及各个环节进行分析，并明确各自角色的定位和职责。

首先，在可靠事件模式中存在一个事件生产者。该事件生产者处于操作的主导地位，它根据业务操作通过事件的方式发送业务操作的结果。在我们的案例中，显然订单服务是事件的生产者。

其次，事件消费者是被动方，负责根据事件来处理自身业务逻辑。显然，在我们的案例中，支付服务属于事件消费者。

有了事件生产者和事件消费者之后，我们将引出最重要的事件服务。这里事件服务的主要作用就是管理本地事件表，它能够存储、确认并发送事件，同时根据不同状态查询事件信息并确定事件已被事件消费者成功消费。事件服务由以下三个组件构成，其中事件确认组件和事件



恢复组件都是用于处理事件没有被正常消费的场景。

### (1) 事件确认组件

事件确认表现为一种定时机制,用于处理事件没有被成功发送的场景。例如,在图 5-19 中,订单服务在完成业务操作之后需要发送事件到本地事件表,如果这个过程中事件没有发送成功,我们就需要对这些事件进行重新发送,我们把这个过程称之为事件确认。

### (2) 事件恢复组件

事件恢复组件同样是一种定时机制,根据本地事件表中的事件状态,专门处理状态为已确认但还没有被成功消费且已超时的事件。基本的事件恢复策略就是向消费者重新发送事件,并在消费者成功消费之后通过更新事件状态的方式将该事件在本地事件表中进行逻辑删除。

### (3) 实时消息传递组件

实时消息传递组件就是基于特定的消息中间件工具和框架将事件作为消息进行发送的组件。目前可供选择的主流消息中间件包括 RabbitMQ、ActiveMQ、RocketMQ、Kafka 等。

结合事件生产者、事件消费者以及事件服务的整体系统架构如图 5-20 所示。我们在图 5-20 中,对各个组件的定位做了描述并添加了组件之间的交互过程。

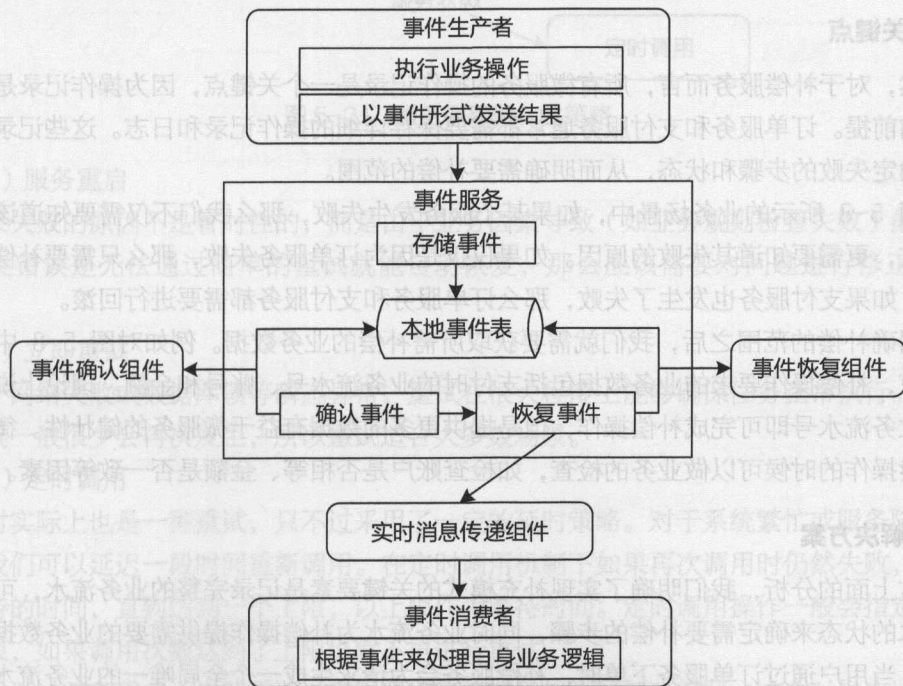


图 5-20 可靠事件模式实现架构图



## 5.2.4 补偿模式

在本节中，我们同样将结合图 5-8 中的业务场景，分析另一种保证数据一致性的常见方法，即基于补偿模式的设计思路以及实现方案。

### 1. 基本思路

补偿模式（Compensation Pattern）的基本思路在于使用一个额外的补偿服务来协调各个需要保证一致性的微服务，补偿服务按顺序依次调用各个微服务，如果某个微服务调用失败就撤销之前所有已经完成的微服务。在这个过程中，补偿服务对需要保证一致性的微服务提供补偿操作。

从图 5-8 所示的业务需求中，我们可以看到需要协调订单服务和支付服务，如果支付服务失败，就需要取消之前的下单服务，该流程中所包含的两个微服务都需要提供补偿操作。

为了降低开发的复杂性和提高效率，补偿服务通常实现为一个通用的补偿框架。补偿框架提供服务编排和自动完成补偿的能力。

### 2. 关键点

显然，对于补偿服务而言，所有微服务的操作记录是一个关键点，因为操作记录是执行取消操作的前提。订单服务和支付服务通常都需要保存详细的操作记录和日志。这些记录和日志有助于确定失败的步骤和状态，从而明确需要补偿的范围。

在图 5-8 所示的业务场景中，如果某个服务发生失败，那么我们不仅需要知道该服务发生了失败，更需要知道其失败的原因。如果仅仅是因为订单服务失败，那么只需要补偿一个服务即可；如果支付服务也发生了失败，那么订单服务和支付服务都需要进行回滚。

在明确补偿的范围之后，我们就需要获取所需补偿的业务数据。例如对图 5-8 中的支付服务而言，补偿操作要求的业务数据包括支付时的业务流水号、账号和金额。理论上说可根据唯一的业务流水号即可完成补偿操作，但是提供更多的数据有益于微服务的健壮性。微服务在收到补偿操作的时候可以做业务的检查，如检查账户是否相等、金额是否一致等因素。

### 3. 解决方案

通过上面的分析，我们明确了实现补偿模式的关键要素是记录完整的业务流水，可以通过业务流水的状态来确定需要补偿的步骤，同时业务流水为补偿操作提供需要的业务数据。具体实现上，当用户通过订单服务下单时，补偿服务会为请求生成一个全局唯一的业务流水号，并在调用各个服务的同时记录完整的状态。例如，如果订单服务调用成功，就记录生成订单的业务流水并更新业务状态；而在调用支付服务时，同样记录支付相关的业务流水并更新业务状态。



一旦发生异常，例如支付发生失败，那么补偿服务在记录支付业务流水状态的同时，也会另外记录一条事件，说明业务出现了异常。接着执行补偿过程，补偿服务可以从业务流水的状态中知道补偿的范围，补偿过程中需要的业务数据同样也可以从记录的业务流水中获取。

这里需要注意一点，补偿服务作为一个服务调用过程同样存在调用不成功的情况。这个时候需要通过一定的健壮性机制来保证补偿的成功率，也就是说补偿服务的相关操作本身就应该具备幂等性。当然，如果只是一味的失败就立即重试会给工作服务造成不必要的压力，我们要根据服务执行失败的原因来选择不同的重试策略。图 5-21 展示了这些健壮性策略，这些策略并不限于补偿场景，同样可以用于其他类似的业务操作。

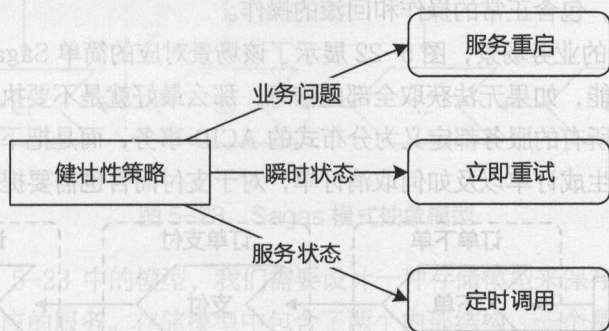


图 5-21 服务调用健壮性策略

### （1）服务重启

如果失败的原因不是暂时性的，而是由于业务因素导致（如业务规则检查失败）的业务错误，这类错误是无法通过简单的重试就能自动恢复，那么应该需要对问题进行修正后重新执行。

### （2）立即重试

对于网络失败或数据库锁等瞬态异常，重试在很大程度上能够确保任务正常执行。因为类似的错误一般很少会再次发生，所以重试适合大多数场景。

### （3）定时调用

定时实际上也是一种重试，只不过采用了一定的延时策略。对于系统繁忙或服务降级的情 况下，我们可以延迟一段时间重新调用。在定时调用机制下如果再次调用时仍然失败，可逐渐增加等待的时间，直到达到一个上限，以上限作为等待时间。定时调用操作一般会指定调用的次数上限，如果调用次数达到了上限也就不再进行重试。

如果通过服务重启、立即重试、定时调用等策略仍然不能解决问题，这个时候应该通过一种手段通知相关人员进行处理，这就是本节最后要讲的人工干预模式。在介绍人工干预模式之前，我们还有 Sagas 长事务模式和 TCC 模式同样可以实现类似补偿模式的效果。



## 5.2.5 Sagas 长事务模式

Sagas 模式属于一个错误管理模式,可以同时用于控制复杂事务的执行和回滚。

### 1. 基本思路

Sagas 最开始是为了实现长时间的本地事务,现在也用于一些跨越多个数据的分布式事务。长时间持续的事务无法简单地通过一些典型的 ACID 模型以及使用多段提交配合持有锁的方式来实现。Sagas 策略正是用来解决这个问题,和多段式分布式事务处理不同,Sagas 会将工作分成单独的事务,包含正常的操作和回滚的操作。

结合图 5-8 所示的业务场景,图 5-22 展示了该场景对应的简单 Sagas 模型。用户需要完成订单下单和支付功能,如果无法获取全部的信息,那么最好就是不要执行下单操作。对开发者来说,我们不是将所有的服务都定义为分布式的 ACID 事务,而是把下单行为定义为一个整体,其中包含如何去生成订单以及如何取消订单,对于支付而言也需要提供同样的逻辑。

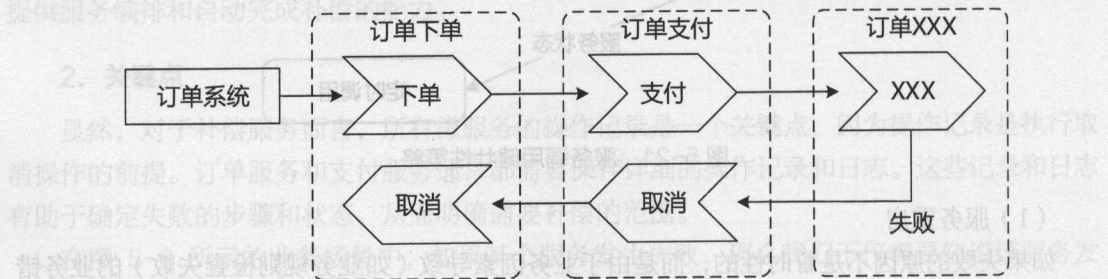


图 5-22 Sagas 模式示意图

在图 5-22 中,我们可以将订单和支付服务组合在一起构成一个服务链。开发者还可以将整个服务链加密,这样只有该服务链的接收者才能够操控这个服务链。当一个服务完成后,会将完成的信息记录到一个集合(比如一个队列)中,之后可以通过这个集合访问到对应的服务。当一个服务失败时,服务本身将本地清理完毕并将消息发送给该集合,从而路由到之前执行成功的服务,然后回滚所有的事务。

### 2. 解决方案

我们可以根据图 5-22 所示的流程抽象出 Sagas 模型的一般表述。在 Sagas 事务模型中,一个长事务是由一个预先定义好执行顺序的子事务集合和它们对应的补偿子事务集合所组成。典型的一个完整的交易由  $T_1$ 、 $T_2$ 、……、 $T_n$  等多个业务活动组成,每个业务活动可以是本地操作或者是远程操作,而每个业务活动都有对应的取消活动  $C_1$ 、 $C_2$ 、……、 $C_n$ 。所有的业务活动在 Sagas 事务下要么全部成功,要么全部回滚,不存在中间状态。从上述对 Sagas 长事



务模型描述来看，我们也可以把它看作是一种补偿模式。

对于一个 Sagas 链路而言，各个业务活动执行过程中都会依赖于上下文（Context），并提供执行（Perform）和取消（Cancel）两个入口。可以使用图 5-23 所示的模型对 Sagas 链路进行抽象和建模。

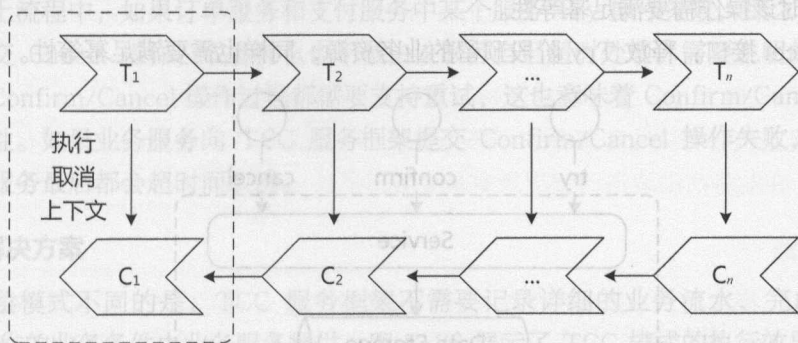


图 5-23 Sagas 模式抽象模型

至于如何实现图 5-23 中的模型，我们需要设计一种存储模型来保存执行上下文并通过该存储模型来索引到对应的服务。存储模型中包含了两个内部结构，一个是完成的任务，一个是等待执行的任务。如果成功就会将任务向前执行，如果失败就会向后执行。实现上的一种思路是采用队列（Queue）和栈（Stack）数据结构，一方面使用队列来向前执行，另一方面使用栈来向后执行。

为了实现 Sagas 模型，每个业务活动都是一个原子操作，而且需要每个业务活动均提供确认和取消操作，当任何一个业务活动发生错误时，按照执行的逆向顺序，实时执行取消操作，进行事务回滚。

如果回滚失败，需要记录失败事务日志，通过重试策略进行重试。重试依然失败的场景，提供定时服务，对回滚失败的业务进行定时修正；针对定时修正依然失败的业务，就只能等待最后的人工干预方式进行最后的修正。这些实现上的详细策略与上一节中介绍的补偿模式一致。

## 5.2.6 TCC 模式

TCC（Try/Confirm/Cancel）模式是对补偿模式的优化，由 Atomikos（<https://www.tomikos.com/>）公司的创始人提出。

### 1. 基本思路

在 TCC 模式中，一个完整的 TCC 业务由一个主服务和若干个从服务组成，主服务发起



并完成整个业务流程。TCC 模式要求从服务提供三个接口：Try、Confirm、Cancel。基于 TCC 模式的服务抽象如图 5-24 所示。

(1) Try 接口：完成所有业务规则检查，预留业务资源。

(2) Confirm 接口：真正执行业务，其自身不作任何业务检查，只使用 Try 阶段预留的业务资源，同时该操作需要满足幂等性。

(3) Cancel 接口：释放 Try 阶段预留的业务资源，同样也需要满足幂等性。

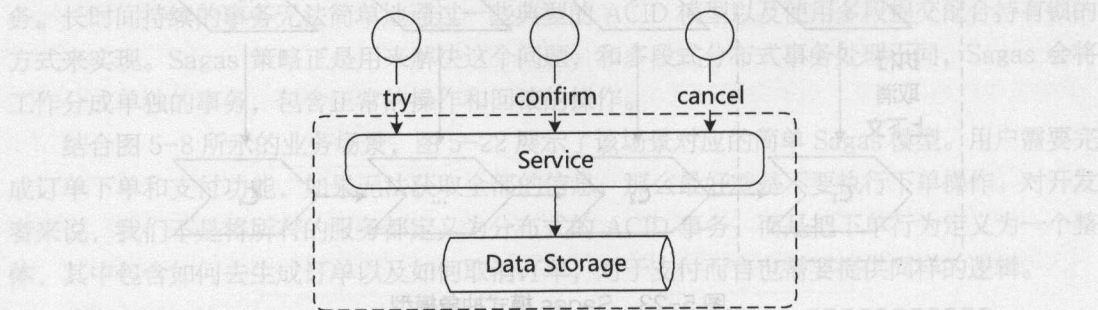


图 5-24 TCC 模式中的服务抽象

我们还是通过订单系统的示例来进一步理解 TCC 模式的设计思想。订单系统拆分成订单下单和订单支付两个场景，使用 TCC 模式执行效果如图 5-25 所示。

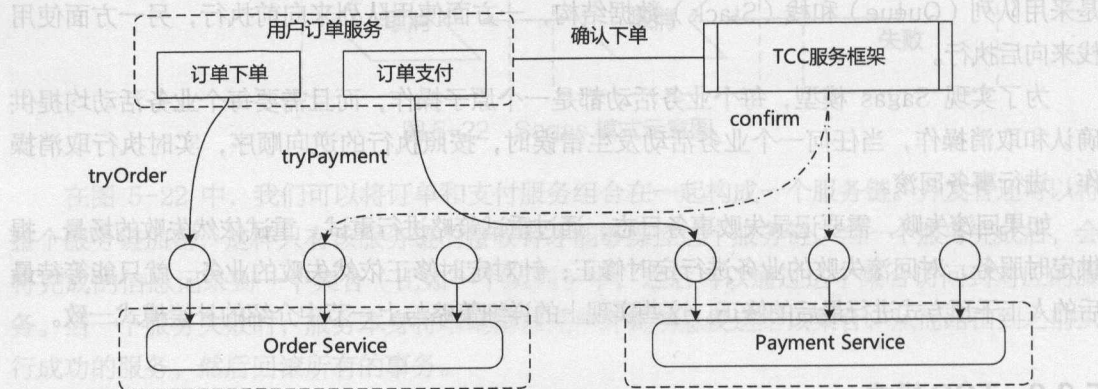


图 5-25 TCC 模式执行效果

(1) Try 阶段：尝试执行业务。在该阶段中，一方面要完成所有业务检查，如针对该次订单下单操作，需要验证商品的可用性以及用户账户中是否有足够的余额；另一方面，也需要预留业务资源，如把用户账户中的部分余额进行冻结用于支付该订单，确保不会出现其他并发进程扣减了账户的余额而导致在后续的真正支付操作过程中，账户可用余额不够的情况。

(2) Confirm 阶段：执行业务。在该阶段中真正执行业务，如果 Try 阶段一切正常，则



执行下单操作并扣除用户账户中的支付金额。

(3) Cancel 阶段：取消执行业务。在该阶段中释放 Try 阶段预留的业务资源，如果 Try 阶段部分成功，如商品可用且正常下单，但账户余额不够而冻结失败，则需要对产品下单做取消操作，释放被占用的该商品。

在以上流程中，如果订单服务和支付服务中某个服务 Try 操作失败，那么可以向 TCC 服务框架提交 Cancel 操作，或者什么也不做由该服务自己超时处理。需要说明的是为保证业务成功率，Confirm/Cancel 操作过程都需要支持重试，这也意味着 Confirm/Cancel 的实现必须具有幂等性。如果业务服务向 TCC 服务框架提交 Confirm/Cancel 操作失败，不会导致不一致，因为服务最后都会超时而取消。

## 2. 解决方案

与补偿模式不同的是，TCC 服务框架不需要记录详细的业务流水，完成 Confirm 和 Cancel 操作的业务条件由业务服务提供。图 5-26 展示了 TCC 模式的执行效果图，可以看到 TCC 模式同样由两个阶段组成。在第一阶段中，主业务服务分别调用所有从业务服务的 Try 操作，并在活动管理器中登记所有从业务服务。当所有从业务服务的 Try 操作都调用成功或者某个从业务服务的 Try 操作失败，进入第二阶段。在第二阶段中，主业务服务根据第一阶段的执行结果来执行 Confirm 或 Cancel 操作。如果第一阶段所有 Try 操作都成功，则主业务服务调用所有从业务活动的 Confirm 操作；否则调用所有从业务服务的 Cancel 操作。

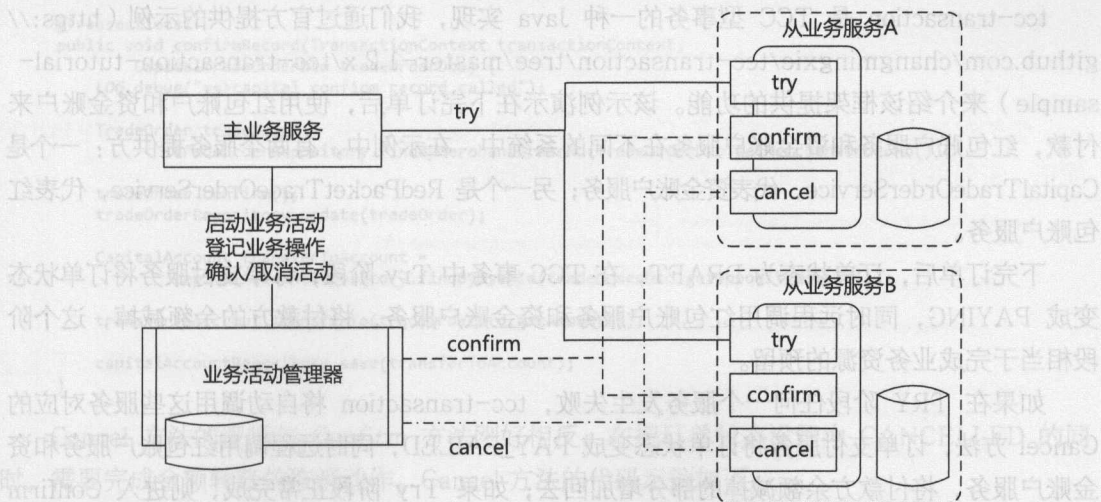


图 5-26 TCC 模式执行效果图

从图 5-26 中可以看到一个完整的 TCC 事务参与方包括三部分。



（1）主业务服务：主业务服务为整个业务活动的发起方，如前面提到的订单处理场景，订单应用系统即是主业务服务。

（2）从业务服务：从业务服务负责提供 TCC 业务操作，是整个业务活动的操作方。从业务服务必须实现 Try、Confirm 和 Cancel 三个接口，供主业务服务调用。由于 Confirm 和 Cancel 操作可能被重复调用，故要求 Confirm 和 Cancel 两个接口必须保证幂等性。前面的订单场景中的订单下单服务和订单支付服务即为从业务服务。

（3）业务活动管理器：业务活动管理器管理控制整个业务活动，包括记录维护 TCC 全局事务的事务状态和每个从业务服务的子事务状态，并在业务活动提交时确认所有从业务服务 Confirm 操作，在业务活动取消时调用所有从业务服务的 Cancel 操作。

### 3. 实现策略

在实现 TCC 模式上，最重要的工作是设计一个稳定的、高可用的、扩展性强的 TCC 事务管理器。当然，设计并实现这样一个 TCC 事务管理器无疑是困难的，我们也不想重复去造轮子，幸好目前业界已经有一些相对比较成熟的实现方案可供参考。例如，tcc-transaction（<https://github.com/changmingxie/tcc-transaction>）、ByteTCC（<https://github.com/iuyangming/ByteTCC>）以及 Atomikos 公司自己的 TCC 方案（<https://www.atomikos.com/Blog/TCCForTransactionManagementAcrossMicroservices>）。这里以 tcc-transaction 为例介绍其实现方法。

tcc-transaction 是 TCC 型事务的一种 Java 实现，我们通过官方提供的示例（<https://github.com/changmingxie/tcc-transaction/tree/master-1.2.x/tcc-transaction-tutorial-sample>）来介绍该框架提供的功能。该示例演示在下完订单后，使用红包账户和资金账户来付款，红包账户服务和资金账户服务在不同的系统中。在示例中，有两个服务提供方：一个是 CapitalTradeOrderService，代表资金账户服务；另一个是 RedPacketTradeOrderService，代表红包账户服务。

下完订单后，订单状态为 DRAFT，在 TCC 事务中 Try 阶段，订单支付服务将订单状态变成 PAYING，同时远程调用红包账户服务和资金账户服务，将付款方的余额减掉，这个阶段相当于完成业务资源的预留。

如果在 TRY 阶段任何一个服务发生失败，tcc-transaction 将自动调用这些服务对应的 Cancel 方法，订单支付服务将订单状态变成 PAY\_FAILED，同时远程调用红包账户服务和资金账户服务，将付款方余额减掉的部分增加回去；如果 Try 阶段正常完成，则进入 Confirm 阶段，订单支付服务将订单状态变成 CONFIRMED，同时远程调用红包账户服务和资金账户服务对应的 Confirm 方法将收款方的余额增加。显然，从业务完整性上讲，在 Confirm 方法和 Cancel 方法中都应该实现幂等性。



以资金账户服务 CapitalTradeOrderService 为例,在对外暴露的主入口上通过添加 tcc-transaction 提供的@Compensable 注解就可以关联 Confirm 方法和 Cancel 方法,其代码示例如下。

```
@Override
@Compensable(confirmMethod = "confirmRecord", cancelMethod = "cancelRecord")
@Transactional
public String record(TransactionContext transactionContext,
    CapitalTradeOrderDto tradeOrderDto) {
    LOG.debug("=>capital try record called");

    TradeOrder tradeOrder = new TradeOrder(
        tradeOrderDto.getSelfUserId(),
        tradeOrderDto.getOppositeUserId(),
        tradeOrderDto.getMerchantOrderNo(),
        tradeOrderDto.getAmount()
    );

    tradeOrderRepository.insert(tradeOrder);

    CapitalAccount transferFromAccount =
        capitalAccountRepository.findById(tradeOrderDto.getSelfUserId());

    transferFromAccount.transferFrom(tradeOrderDto.getAmount());

    capitalAccountRepository.save(transferFromAccount);
    return "success";
}
```

其中,在 Confirm 方法中,我们发现一方面将订单状态更新为 CONFIRMED,另一方面则完成金额转账工作,Confirm 方法的代码示例如下。

```
@Transactional
public void confirmRecord(TransactionContext transactionContext,
    CapitalTradeOrderDto tradeOrderDto) {
    LOG.debug("=>capital confirm record called");

    TradeOrder tradeOrder =
        tradeOrderRepository.findByMerchantOrderNo(tradeOrderDto.getMerchantOrderNo());

    tradeOrder.confirm();
    tradeOrderRepository.update(tradeOrder);

    CapitalAccount transferToAccount =
        capitalAccountRepository.findById(tradeOrderDto.getOppositeUserId());

    transferToAccount.transferTo(tradeOrderDto.getAmount());

    capitalAccountRepository.save(transferToAccount);
}
```

Cancel 方法的逻辑与 Confirm 方法刚好相反,在把订单状态设置为 CANCELLED 的同时,需要完成金额转账的取消动作。Cancel 方法的代码示例如下。



```
@Transactional
public void cancelRecord(TransactionContext transactionContext,
    CapitalTradeOrderDto tradeOrderDto) {
    LOG.debug("=>capital cancel record called");

    TradeOrder tradeOrder =
        tradeOrderRepository.findByMerchantOrderNo(tradeOrderDto.getMerchantOrderNo());

    if(null != tradeOrder && "DRAFT".equals(tradeOrder.getStatus())) {
        tradeOrder.cancel();
        tradeOrderRepository.update(tradeOrder);

        CapitalAccount capitalAccount =
            capitalAccountRepository.findById(tradeOrderDto.getSelfUserId());

        capitalAccount.cancelTransfer(tradeOrderDto.getAmount());

        capitalAccountRepository.save(capitalAccount);
    }
}
```

在资金账户服务 CapitalTradeOrderService 和红包账户服务 RedPacketTradeOrderService 的上层，支付服务会统一调用这两个服务来完成支付操作，其示例代码如下。

```
@Compensable(confirmMethod = "confirmMakePayment", cancelMethod = "cancelMakePayment")
@Transactional
public void makePayment(Order order, BigDecimal redPacketPayAmount,
    BigDecimal capitalPayAmount) {

    order.pay(redPacketPayAmount, capitalPayAmount);
    orderRepository.updateOrder(order);

    // 资金账户交易订单记录
    capitalTradeOrderService.record(null, buildCapitalTradeOrderDto(order));

    // 红包账户交易订单记录
    redPacketTradeOrderService.record(null, buildRedPacketTradeOrderDto(order));
}
```

我们看到这该方法中也实现了 Confirm 方法和 Cancel 方法，这两个方法的实现逻辑比较简单，主要用于管理订单的统一状态，其代码示例如下。

```
public void confirmMakePayment(Order order,
    BigDecimal redPacketPayAmount, BigDecimal capitalPayAmount) {

    order.confirm();

    orderRepository.updateOrder(order);
}

public void cancelMakePayment(Order order,
    BigDecimal redPacketPayAmount, BigDecimal capitalPayAmount) {

    order.cancelPayment();

    orderRepository.updateOrder(order);
}
```

而在该支付服务的外围才是对外的面向前端的访问入口，其代码示例如下。



```
public String placeOrder(long payerUserId, long shopId,
    List<Pair<Long, Integer>> productQuantities,
    BigDecimal redPacketPayAmount) {
    Shop shop = shopRepository.findById(shopId);
    Order order = orderService.createOrder(payerUserId,
        shop.getOwnerUserId(), productQuantities);
    try {
        // 付款
        paymentService.makePayment(order, redPacketPayAmount,
            order.getTotalAmount().subtract(redPacketPayAmount));
    } catch (ConfirmingException confirmingException) {
    } catch (CancellingException cancellingException) {
    } catch (Throwable e) {
    }
    return order.getMerchantOrderNo();
}
```

至此, 我们完成了整个业务流程的闭环管理, 而其中用到了两层的 TCC 事务处理机制, 整个业务流程的时序图参考图 5-27。

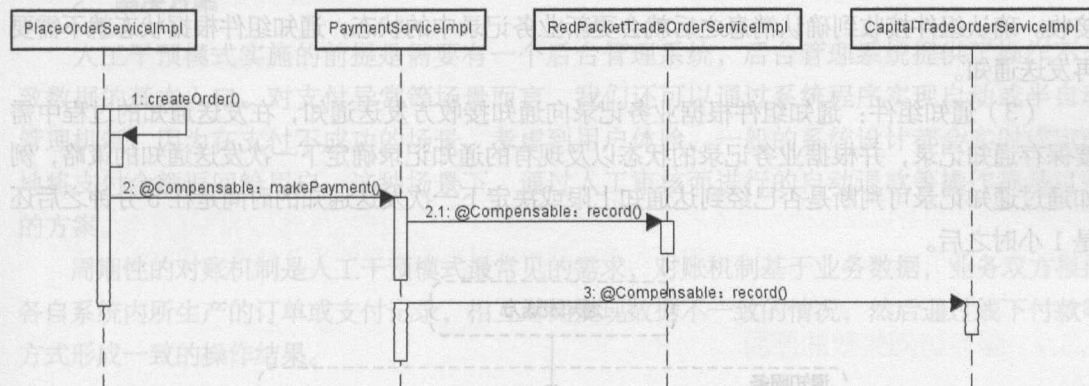


图 5-27 基于 tcc-transaction 的下单支付流程时序图

## 5.2.7 最大努力通知模式

与补偿模式不同, 最大努力通知模式本质上是一种通知类的实现方案。

### 1. 基本思路

最大努力通知模式的基本思路是通知发送方在完成业务处理后向通知接收方发送通知消息。



因为这种通知消息允许丢失，所以当消息的接收方没有成功消费消息时，消息的发送方需要进行重复发送直到消费者消费成功或达到某种发送终止条件。一般，消息发送方可以设置复杂的通知规则，如采用 15s、3m、10m、30m、1h、2h、6h、15h 等阶梯式时间的通知方式。

另一方面，在最大努力通知模式中，通知接收方也可以使用主动方所提供的查询和对账接口获取数据，用于恢复通知失败所导致的业务数据。

最大努力通知模式在通知类场景应用广泛，以支付宝为例，通过回调商户提供的回调接口，通过多次通知、查询对账等手段完成交易业务平台间的商户通知。

## 2. 解决方案

最大努力通知模式比较容易理解，实现上也比较简单。其基本的系统结构参考图 5-28 中的通知服务，包括如下组件。

（1）查询组件：通知发送方处理业务并把业务记录保存起来，查询组件提供查询入口供通知接收方主动查询业务数据，避免数据丢失。

（2）确认组件：当通知接收方成功接收到通知时，需要与通知发送方确认通知已被正常接收。确认组件接收到确认消息之后就会更新业务记录中的状态，通知组件根据状态就不需要再发送通知。

（3）通知组件：通知组件根据业务记录向通知接收方发送通知，在发送通知的过程中需要保存通知记录，并根据业务记录的状态以及现有的通知记录确定下一次发送通知的策略，例如通过通知记录可判断是否已经到达通知上限或决定下一次发送通知的时间是在 5 分钟之后还是 1 小时之后。

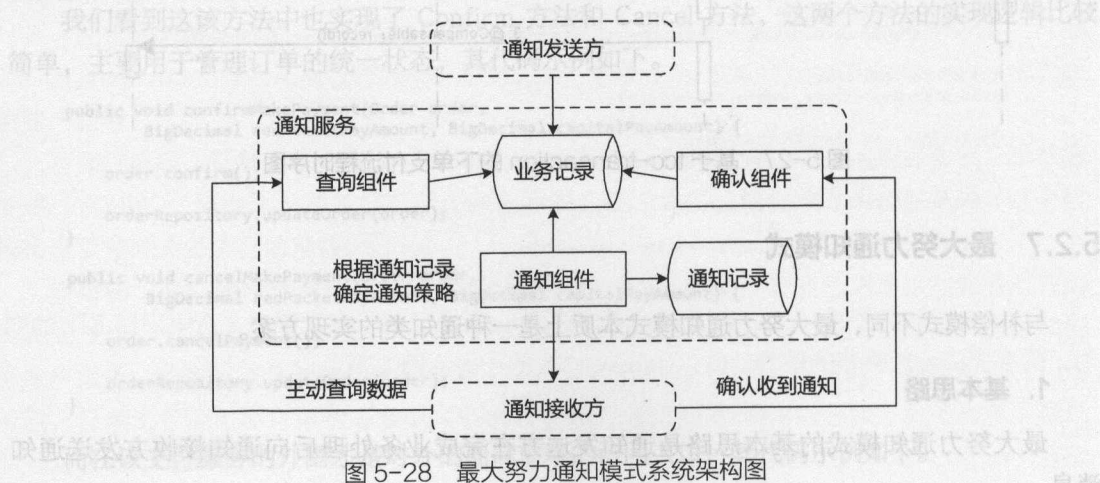


图 5-28 最大努力通知模式系统架构图



最大努力通知模式适合于对业务最终一致性的时间敏感度比较低的场景，一般用于类似支付宝与商户集成类跨企业的业务活动。

### 5.2.8 人工干预模式

人工干预模式严格意义上并不是一种数据一致性的实现机制，当前面介绍的各种模式都无法满足需要时，人工干预模式更多地是提供一种替代方案。

#### 1. 基本思路

如果有些业务由于瞬时的网络故障或调用超时等问题，通过前面所介绍的可靠事件模式、补偿模式等方法一般都能得到很好的解决。但是系统中的很多微服务有时候需要依赖于外部系统的可用性，在一些重要的业务场景下还需要通过人工干预的方式来保证真正的一致性。典型的场景就是图 5-8 所示的支付服务和第三方的支付宝系统之间可能会有定期对账的过程。

#### 2. 解决方案

人工干预模式实施的前提是需要有一个后台管理系统，后台管理系统提供了操作不一致数据的基本入口。对支付异常等场景而言，我们还可以通过系统程序实现自动或半自动管理机制，因为在支付不成功的场景，考虑到用户体验，一般的系统设计都会实时或定时地将支付金额返回给用户。这种场景下，通过人工审核而进行的自动退款等操作都是可行的方案。

周期性的对账机制是人工干预模式最常见的需求，对账机制基于业务数据，业务双方根据各自系统内所生产的订单或支付记录，相互对比发现数据不一致的情况，然后通过线下付款等方式形成一致的操作结果。

### 5.2.9 数据一致性模式总结

在日常系统设计和实现过程中，需要根据业务需求的特点对选择何种最终一致性的实现方案做出决策。对于基于金融、支付等业务体系，数据一致性要求极高，需要保证严格的实时一致性要求；而对于基于社交类的应用场景，可以采用局部实时一致、最终全局一致的实现思路。在本书所着重介绍的移动医疗系统中，预约挂号等场景需要保证较高的实时性，而对于检查检验报告查询类的业务场景，通常时间都不会是太大的约束条件。表 5-2 是对以上所有在分布式环境下实现数据最终一致性模式的总结，可以直接参考该表中的对比并结合自身业务需求，选择合适的数据一致性实现模型。



表 5-2

数据一致性模式总结

	一致性	吞吐量	复杂度
2PC	强一致性	低	易
3PC	强一致性	低	易
可靠事件模式	最终一致性	中	中
补偿模式	最终一致性	高	难
Sagas 模式	最终一致性	高	难
TCC 模式	最终一致性	中	难
最大努力通知模式	最终一致性	高	中

作为系统分析和设计人员，我们都应该有一种“兜底”思维，即不管实现方案是否完美，最后都要有一个备选方案，这种备选方案可能并不一定满足日常的业务场景，但当出现异常情况时，我们还是可以通过该备选完成正常业务的闭环。针对表 5-2 中的所有一致性实现模式，我们认为上一节中介绍的人工干预模式就是这样一种兜底方案。

## 5.3 服务可靠性

在微服务架构中，各个服务独立部署且服务与服务之间存在相互依赖关系。和单块系统相比，微服务架构中出现服务访问失败的原因和场景非常复杂，这就需要我们z从服务可靠性的角度出发对服务自身以及服务与服务之间的交互过程进行设计。服务可靠性是微服务架构的关键要素之一，本节从服务访问失败的原因入手，分析确保服务可靠性的各种技术和方法。

### 5.3.1 服务访问失败的原因

在讨论服务可靠性之前，我们有必要深入分析服务访问失败的原因并给出分布式环境中典型的雪崩现象的产生原因。

#### 1. 服务访问失败原因分类

对于分布式环境中的服务而言，服务访问失败的原因主要包括以下几类。

##### (1) 硬件失败

硬件失败不常见，但一旦出现问题通常都是灾难性。这里的硬件失败不仅包括机房失火、机器损坏等不可抗力所导致的、发生概率极低的硬件不可用，更多的是指由于日志文件过大导致硬盘无法写入、网络路由无效等可以通过调整硬件状态进行恢复的失败场景。



## （2）分布式环境的固有原因

我们已经在本书 1.1.2 节中介绍了分布式系统的基本特征，明确在分布式系统中由于网络传输的三态性、异构系统集成等因素会导致远程过程调用发生异常情况。微服务架构作为分布式系统的一种延伸，同样也存在类似的问题。这些问题构成了服务访问失败的原因，我们无法完全消除这些原因，只能在设计和实现时加以预防，以及在发生时降低其所造成的影响。

## （3）服务自身失败

微服务本身当然也可能发生失败，由于设计实现上考虑不周、代码中存在的缺陷等因素所造成的服务发生失败的场景也不少见，这些场景需要我们深入分析并找到解决问题的方法。

## （4）服务依赖失败

除了服务自身失败，服务访问失败的另一种表现形式是服务依赖失败。服务依赖失败较之服务自身失败而言其影响更大，也更加难以发现和处理。服务依赖失败是我们在设计微服务架构中所需要重点考虑的失败原因，因为服务依赖失败会造成失败扩散，从而形成服务访问的雪崩效应。

以上服务访问失败的原因中，雪崩效应是微服务架构需要重点考虑的一个主题，接下来我们将对这个主题做进一步展开。

## 2. 服务访问雪崩效应

图 5-29 展示了雪崩效应的效果图（边框为虚线的服务代表不可用），可以看到 A、B、C、D、E 共 5 个服务存在依赖关系，A 为服务提供者，B 为 A 的服务调用者，C、D 和 E 是 B 的服务调用者。假如 A 变成不可用，就会引起 B 的不可用，并将不可用逐渐扩散到 C、D 和 E 时，就造成了整个服务体系发生雪崩。

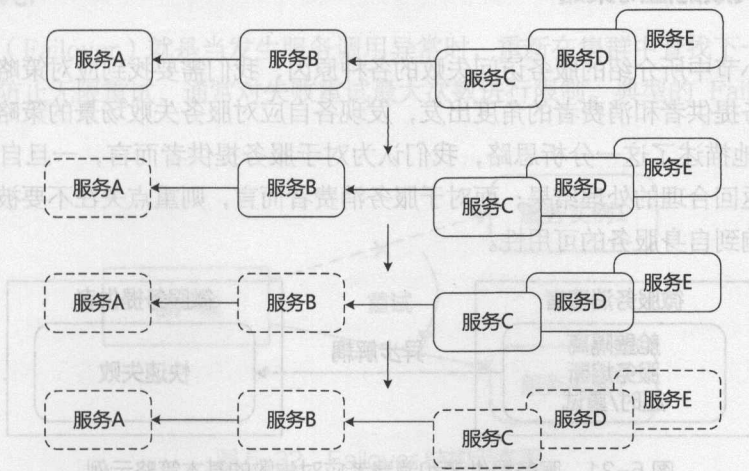


图 5-29 服务雪崩效应效果图



服务雪崩效应的产生是一种扩散效应，我们可以对图 5-29 中的现象进行剥离，先从 A 和 B 两个服务之间的交互展开讨论（见图 5-30，边框为虚线的服务代表不可用）。我们知道从角色出发，服务可以分为提供者（Provider）和消费者（Consumer），如图 5-30 所示的 A 服务就是提供者，而 B 服务就是消费者。图 5-30 也展示了雪崩效应的产生的三个阶段，即首先提供者 A 服务发生不可用，然后消费者 B 服务重试加大流量，最后导致 B 服务自身也不可用。

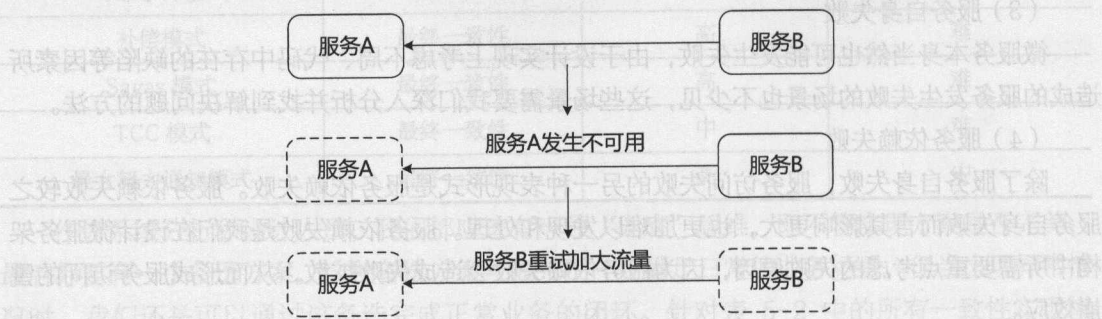


图 5-30 两个服务交互图

服务 A 不可用的原因有很多，包括前面提到的服务器硬件等环境问题、服务自身存在 Bug 等因素。而 B 服务通过用户不断提交服务请求等手工重试或代码逻辑自动重试等手段会进一步加大对 A 服务的访问流量。因为 B 服务使用同步调用，会产生大量的等待线程占用系统资源。一旦线程资源被耗尽，B 服务提供的服务本身也将处于不可用状态，从而产生如图 5-30 所示的效果。这一效果在整个服务访问链路上进行扩散，就形成了雪崩效应。

### 5.3.2 服务失败的应对策略

针对上一小节中所介绍的服务访问失败的各种原因，我们需要找到应对策略。我们的思路是分别站在服务提供者和消费者的角度出发，发现各自应对服务失败场景的策略和实现方法。图 5-31 简单地描述了这一分析思路，我们认为对于服务提供者而言，一旦自身服务发生错误，应该快速返回合理的处理结果；而对于服务消费者而言，则重点关注不要被服务提供者所产生的错误影响到自身服务的可用性。

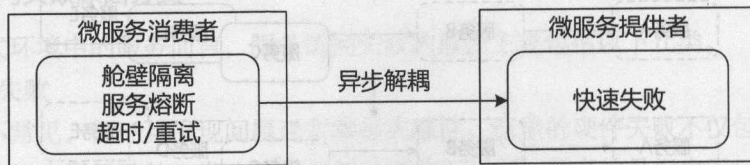


图 5-31 服务提供者和消费者应对失败的基本策略示例



围绕图 5-31, 有些应对策略比较容易想到和实现, 如超时 (Timeout)、重试 (Retry) 和异步解耦。

对于服务消费者而言, 为了保护自身服务的可用性, 可以使用超时机制降低它所依赖服务对其造成的影响。超时机制指的是调用服务的操作可以配置为执行超时, 如果服务未能在这个时间内响应, 将回复一个失败消息。这种策略有利有弊, 因为可能会导致许多并发请求同一个操作一直被阻塞, 直到到达超时时间。设置较短的超时时间有助于解决这个问题。

同时, 为了降低网络瞬态异常所造成的网络通信问题, 可以使用重试机制。正如雪崩效应中的分析结果, 频繁重试也会产生同步等待, 因此合理限制重试次数是一般的做法。

从降低系统耦合度的角度出发, 我们也会发现通过使用一些中间件系统实现服务提供者和服务消费者之间异步解耦, 也能把服务依赖失败的影响分摊到中间件上, 从而降低服务失败的概率。

除此之外, 业界还存在一些更为系统的方法和机制确保服务的可靠性, 包括服务容错、服务隔离、服务限流和服务降级。

### 5.3.3 服务容错

当服务运行在一个集群中, 出现通信链路故障、服务端超时以及业务异常等场景都会导致服务调用失败。容错 (Fault Tolerance) 机制的基本思想是冗余和重试, 即当一个服务器出现问题时不妨试试其他服务器。集群的建立已经满足冗余的条件, 而围绕如何进行重试就产生了几种常见的集群容错策略。

#### 1. Failover

失效转移 (Failover) 就是当发生服务调用异常时, 重新在集群中查找下一个可用的服务提供者。为了防止无限重试, 通常对失败重试最大次数进行限制。典型的 Failover 结构如图 5-32 所示。

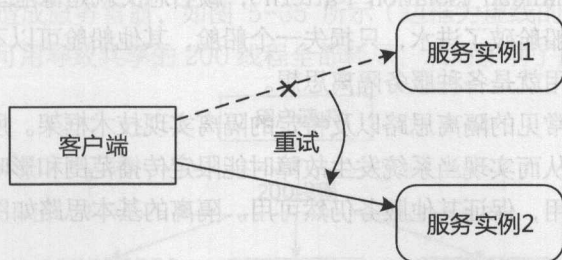


图 5-32 Failover 结构示意图



## 2. Failback

Failback 可以理解为失败通知，当服务调用失败直接将远程调用异常通知给消费者，由消费者捕获异常进行后续处理。

## 3. Failsafe

失败安全（Failsafe）策略中，当获取服务调用异常时，直接忽略。通常将异常写入审计日志等媒介，确保后续可以根据日志记录找到引起异常的原因并解决。该策略可以理解作为一种简单的熔断机制（Circuit Breaker），为了调用链路的完整性，在非关键环节中允许出现错误而不中断整个调用链路。

## 4. Failfast

快速失败（Failfast）策略在获取服务调用异常时，立即报错。显然，Failfast 已经彻底放弃了重试机制，等同于没有容错。在特定场景中，可以使用该策略确保非核心业务服务只调用一次，为重要的核心服务节约宝贵时间。

## 5. Forking

使用分支机制（Forking）时会并行调用多个服务器，只要一个成功即返回；通常用于实时性要求较高的读操作，但需要浪费更多服务资源。

## 6. Broadcast

广播机制（Broadcast）就是逐个调用所有提供者，任意一台报错则报错。通常用于通知所有提供者更新缓存或日志等本地资源信息的业务场景，而不是简单的远程调用。

### 5.3.4 服务隔离

舱壁隔离模式（Bulkhead Isolation Pattern），顾名思义就是像舱壁一样对资源或失败单元进行隔离。如果一个船舱破了进水，只损失一个船舱，其他船舱可以不受影响。舱壁隔离模式在微服务架构中的应用就是各种服务隔离思想。

服务隔离包括一些常见的隔离思路以及特定的隔离实现技术框架。所谓隔离，本质上是对系统或资源进行分割，从而实现当系统发生故障时能限定传播范围和影响范围，即发生故障后只有出问题的服务不可用，保证其他服务仍然可用。隔离的基本思路如图 5-33 所示。

图 5-31 服务隔离示意图



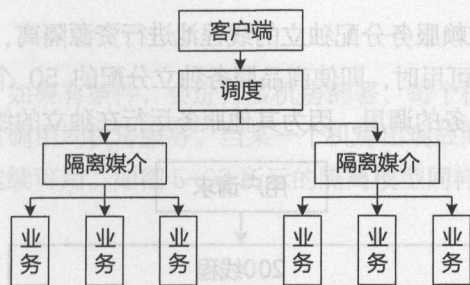


图 5-33 隔离的基本思路

## 1. 线程隔离

线程隔离主要通过线程池（Thread Pool）进行隔离。在实际使用时，我们会把业务进行分类并交给不同的线程池进行处理，当某个线程池处理一种业务请求发生问题时，不会将故障扩散到其他线程池，也就不会影响到其他线程池中所运行的业务，从而保证其他服务可用。在图 5-33 中，我们把隔离媒介替换成线程池就能起到线程隔离的效果。

线程隔离是实现服务隔离的基础，我们通过一个实例来进一步介绍它的工作场景。假如系统存在商品服务、用户服务和订单服务 3 个微服务，然后通过设置运行时环境得到这 3 个服务一共使用 200 个线程，客户端调用这 3 个服务会共享线程池，如图 5-34 所示。

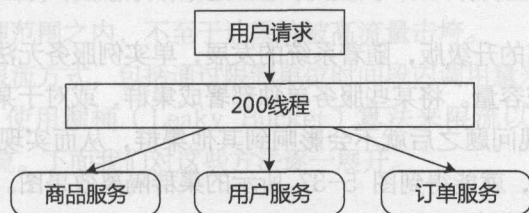


图 5-34 没有使用线程池隔离的场景

在图 5-34 中，如果其中的商品服务不可用，就会出现线程池里所有线程都因同步等待响应而被阻塞，从而造成服务雪崩，如图 5-35 所示（边框为虚线的服务代表不可用），可以看到因为商品服务不可用导致共享的 200 线程全部耗尽，从而影响了用户服务和订单服务。

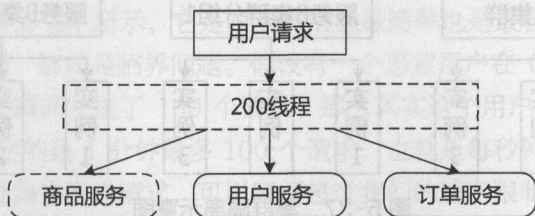


图 5-35 没有使用线程池隔离造成的服务雪崩场景



线程隔离机制将每个依赖服务分配独立的线程池进行资源隔离,从而避免服务雪崩。如图 5-36 所示,当商品服务不可用时,即使商品服务独立分配的 50 个线程全部处于同步等待状态,也不会影响其他依赖服务的调用,因为其他服务运行在独立的线程池中。

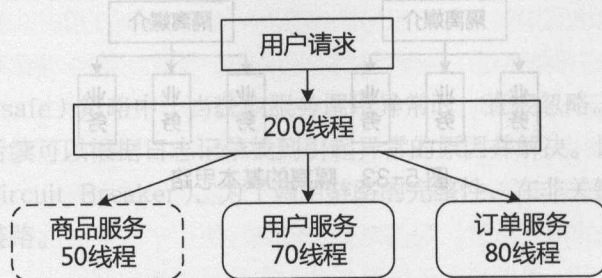


图 5-36 使用线程池隔离的场景

## 2. 进程隔离

进程隔离比较好理解,就是将系统拆分为多个子系统来实现物理隔离,各个子系统运行在独立的容器和 JVM 中,通过进程隔离使得某一个子系统出现问题不会影响到其他子系统。在图 5-33 中,我们同样可以把隔离媒介简单替换成 JVM,就能起到进程隔离的效果。

## 3. 集群隔离

集群隔离是进程隔离的升级版,随着系统的发展,单实例服务无法满足需求了,此时需要使用集群机制来提升系统容量。将某些服务单独部署成集群,或对于某些服务可以进行分组集群管理,某一个集群出现问题之后就不会影响到其他集群,从而实现了故障隔离。我们对图 5-33 所示进行简单改造,就能得到图 5-37 所示的集群隔离效果图。该图中包括了单一的服务 A 集群,也包括分组的服务 B 集群。

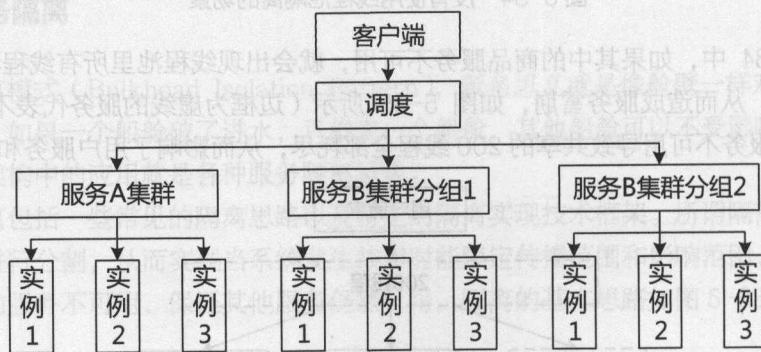


图 5-37 集群隔离示意图



#### 4. 机房隔离

对于大型高可用系统，如果有条件，会进行多机房部署，每个机房的服务都有自己的服务分组，本机房的服务应该只调用同机房服务。当某一个机房出现故障时，可以将请求快速切到另一个机房从而确保服务继续可用。如图 5-33 所示的隔离模型同样适用于机房隔离。

#### 5. 读写隔离

读写隔离也是常见的隔离技术，当用于读取操作的服务器出现故障时，写服务器照常可以运作，反之也是一样。对于离线分析类的应用场景而言，读写隔离可以很好地控制读取操作可能形成的瓶颈对写入操作造成的影响。实际运用中，读写隔离可以有不同的表现形式。例如，当使用 MongoDB 进行海量数据存储时，可以采用热库和存档库概念，将新写入的数据同时存储到热库和存档库。热库只存储最近一个月的数据，而存档库则保存所有的历史数据。当我们想对历史数据进行离线处理时，可以确保热库不受影响。

### 5.3.5 服务限流

为了保证在业务高峰期线上系统也能具备一定的弹性和稳定性，限流就是最常采用的方案之一。所谓限流即流量限制，限流的目的是在遇到流量高峰期或者流量突增时，把流量速率限制在系统所能接受的合理范围之内，不至于让系统被高流量击垮。

目前有几种常见的限流方式，包括通过限制单位时间段内调用量来限流、通过限制系统的并发调用程度来限流、使用漏桶（Leaky Bucket）算法来限流以及使用令牌桶（Token Bucket）算法来进行限流。下面我们对这些方法逐一展开。

#### 1. 计数器法

对于通过限制某个服务在单位时间内的调用量来进行限流的方法，我们需要做的就是使用一个计数器（Counter）统计单位时间段某个服务的访问量，如果超过了我们设定的阈值，则该单位时间段内不允许服务继续响应请求，或者把接下来的请求放入队列中等待到下一个单位时间段继续访问。这里，计数器需要在进入下一个单位时间段时先重置清零。

计数器法示意图如图 5-38 所示，它是限流算法里最简单也是最容易实现的一种算法，但是有一个十分致命的问题，那就是临界问题。假设有一个恶意用户在 0:59 时瞬间发送了 100 个请求；并且在 1:00 又瞬间发送了 100 个请求，那么其实这个用户在 1 秒里面，瞬间发送了 200 个请求。我们规定的是 1 分钟最多 100 个请求，也就是每秒钟最多 1.7 个请求，通过在时间窗口的重置节点处集中发送请求，可以瞬间超过我们的速率限制。用户有可能通过算法的这个漏洞，瞬间压垮我们的应用。



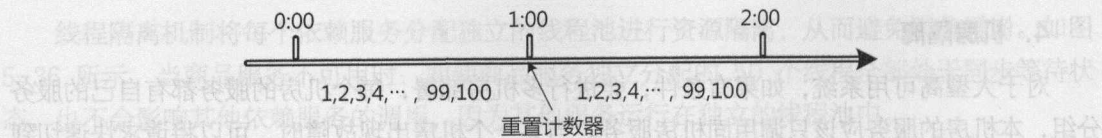


图 5-38 计数器法示意图

## 2. 滑动窗口法

滑动窗口 (Rolling Window) 的工作原理如图 5-39 所示, 在我们的例子中, 一个时间窗口就是 1 分钟。然后将时间窗口进行划分。例如, 图 5-39 中我们就将滑动窗口划成了 6 格, 所以每格代表的是 10 秒钟。每过 10 秒钟, 我们的时间窗口就会往右滑动一格。每一个格子都有自己独立的计数器。例如, 当一个请求在 0:35 秒的时候到达, 那么 0:30~0:39 秒对应的计数器就会加 1。

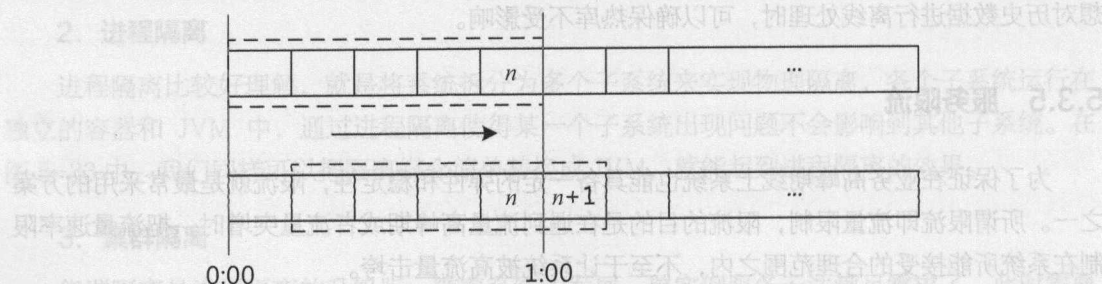


图 5-39 滑动窗口示意图

那么滑动窗口怎么解决刚才的临界问题的呢? 我们可以看到在图 5-39 中, 0:59 到达的 100 个请求会落在  $n$  号格子中, 而 1:00 到达的请求会落在  $n+1$  号格子中。当时间到达 1:00 时, 我们的窗口会往右移动一格, 那么此时时间窗口内的总请求数量一共是 200 个, 超过了限定的 100 个, 所以此时能够检测出来触发了限流。

实际上计数器算法可以认为是滑动窗口算法的一种特例, 只是它没有对时间窗口做进一步划分。由此可见, 当滑动窗口的格子划分得越多, 那么滑动窗口的滚动就越平滑, 限流的统计就会越精确。

## 3. 漏桶算法

漏桶算法是网络中流量整形的常用算法之一, 它有点像我们生活中用到的漏斗, 液体倒进去以后, 总是从下端的小口中以固定速率流出。漏桶算法也有类似的效果, 不管突然流量有多大, 漏桶都保证了流量的常速率输出。类比于服务调用量, 不管服务调用多么不稳定, 我们只固定进行服务输出, 比如每 10 毫秒接受一次服务调用。既然是一个桶, 那就肯定有容量, 由于调用的消费速率已经固定, 那么当桶的容量堆满时服务请求就只能丢弃, 如图 5-40 所示。



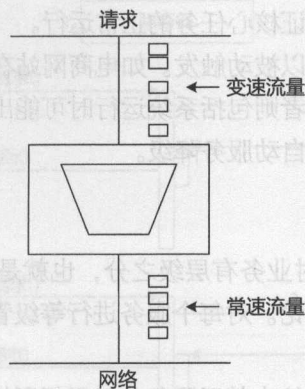


图 5-40 漏桶算法示意图

#### 4. 令牌桶算法

很多应用场景中，除了要求能够限制数据的平均传输速率外，还要求允许某种程度的突发传输。这时候漏桶算法可能就不合适了，令牌桶算法更为适合，令牌桶算法从某种程度上来说是漏桶算法的一种改进。如图 5-41 所示，令牌桶算法的原理是系统会以一个恒定的速度往桶里放入令牌，而如果请求需要被处理，则需要先从桶里获取一个令牌，当桶里没有令牌可取时，则拒绝服务。

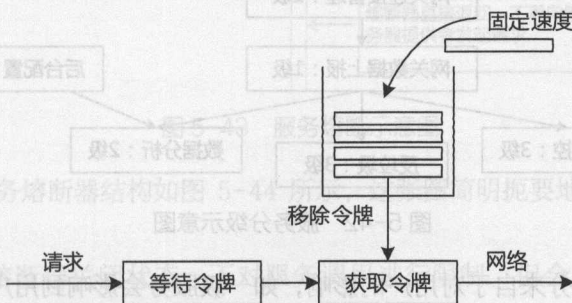


图 5-41 令牌桶算法示意图

漏桶算法与令牌桶算法在表面看起来类似，很容易将两者混淆。但事实上，这两者具有截然不同的特性，且应用于不同场景。漏桶算法与令牌桶算法的主要区别在于漏桶算法能够强行限制数据的传输速率，而令牌桶算法能够在限制数据的平均传输速率的同时还允许某种程度的突发传输。

#### 5.3.6 服务降级

服务降级指的是在服务器压力剧增的情况下，根据当前业务情况及流量对一些服务有策略



地降级，以此释放服务器资源以保证核心任务的正常运行。

降级可以有计划地执行，也可以被动触发。如电商网站在“双十一”期间对部分非核心业务进行手工降级就属于前者，而后者则包括系统运行时可能出现的各种异常情况，为了控制异常的影响范围可以在程序级别实现自动服务降级。

## 1. 服务分级

服务降级在实现上一般需要对业务有层级之分，也就是需要实现服务分级，具体的服务分级方法可以参考 2.2.2 节中的讨论。对每个服务进行等级管理之后，降级一般是从最外围、等级最低的服务开始。

图 5-42 展示了在移动医疗系统中基于服务分级思想所构建的无线网关业务，无线网关用于上传来自无线网络的数据并进行相应的处理。对于这样一个网关系统而言，网关的链接管理和数据上报是优先级最高的业务功能，因为缺少这些功能，网关也就不能正常工作。数据上报之后，对数据进行分析是收集网关数据的目的所在，但数据分析服务可以离线处理，并不一定需要保证服务 100% 的在线率。其他诸如服务监控、反垃圾和后台配置则属于辅助性功能，相应的优先级也就最低。

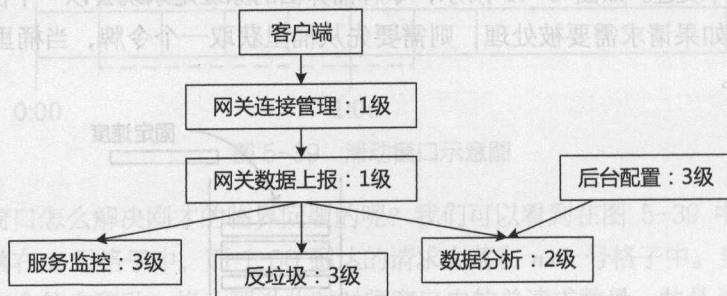


图 5-42 服务分级示意图

以上三级服务的拆分来自于对用户的影响，如一级服务会影响到用户对网关的直接使用；二级服务不影响用户上网，但可能会丢失重要数据，并影响体验；三级服务所对应系统的稳定性则对线上业务无影响。

## 2. 服务熔断

关于服务降级还有一个类似的概念称为服务熔断（Circuit Breaker），服务熔断类似现实世界中的“保险丝”，当某个异常条件被触发，直接熔断整个服务，而不是一直等到此服务超时（见图 5-43）。而服务降级就是当某个服务熔断之后，服务端准备一个本地的回退（Fallback）回调，返回一个缺省值。



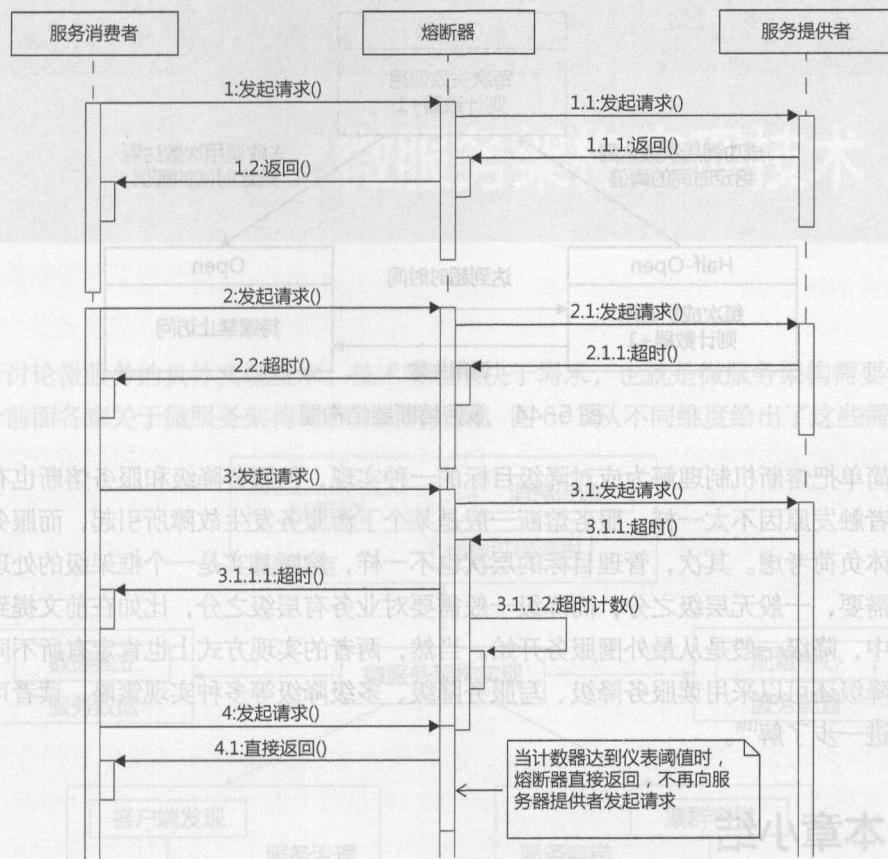


图 5-43 服务熔断示意图

一个最基本的服务熔断器结构如图 5-44 所示，这张图简明扼要地给出了熔断器实现上的三个状态机<sup>[18]</sup>。

(1) Closed：熔断器关闭状态，不对服务调用进行限制，但会对调用失败次数进行积累，到了阈值或一定比例时则启动熔断机制。

(2) Open：熔断器打开状态，此时对下游的调用内部直接返回错误，不走真正的网络调用。同时，熔断器设计了一个时钟选项，当时钟达到了一定时间（这个时间一般设置成平均故障处理时间，也就是 MTTR），进入半熔断状态。

(3) Half-Open：半熔断状态，允许定量的服务请求，如果调用都成功或达到一定比例则认为调用链路已恢复，关闭熔断器；否则认为调用链路仍然存在问题，又回到熔断器打开状态。



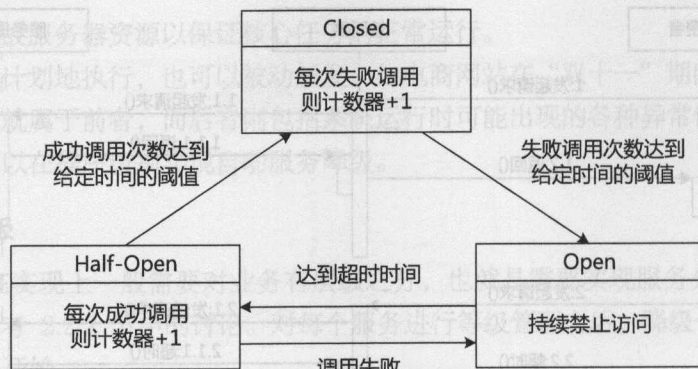


图 5-44 服务熔断器结构图

可以简单把熔断机制理解为应对降级目标的一种实现，但服务降级和服务熔断也有区别。首先，两者触发原因不太一样，服务熔断一般是某个下游服务发生故障所引起，而服务降级一般是从整体负荷考虑。其次，管理目标的层次也不一样，熔断其实是一个框架级的处理，每个微服务都需要，一般无层级之分；而降级一般需要对业务有层级之分，比如在前文提到的服务分级管理中，降级一般是从最外围服务开始。当然，两者的实现方式上也肯定有所不同。

关于降级还可以采用读服务降级、写服务降级、多级降级等多种实现策略，读者可以参考相关资料进一步了解<sup>[19]</sup>。

## 5.4 本章小结

作为微服务架构的重要组成部分，本章详细讨论了服务治理、数据一致性和服务可靠性等三个关键要素。

在服务数量大、依赖关系复杂的情况下，以服务注册中心为代表的服务注册和发现机制是构建微服务架构服务治理体系的基础。关于服务治理，我们还分别从服务提供者和消费者的角度出发讨论服务发布、注册、发现和调用流程，并简要介绍了服务监控的实现思路。

数据一致性在分布式环境中是不得不面对的一个问题，本章针对这个主题给出了目前业界主流的几种实现数据最终一致性的方法，包括可靠事件模式、补偿模式、Sagas 长事务模式、TCC 模式、最大努力通知模式以及人工干预模式。

最后本章从服务访问失败的原因出发，针对如何实现服务可靠性提供了应对方法，并着重介绍了服务容错、服务隔离、服务限流和服务降级等关键策略的实现思路。

本章的内容属于理论体系，关于如何实现这些理论体系背后的设计思路，我们需要给出相应的技术体系、实现工具和工程实践，这些内容将是下一章的重点。



# 微服务架构实现技术

本章讨论微服务的具体实现技术。技术实现取决于需求，也就是微服务架构需要考虑的问题。结合前面各章关于微服务架构各个方面的介绍，图 6-1 从不同维度给出了这些需求。

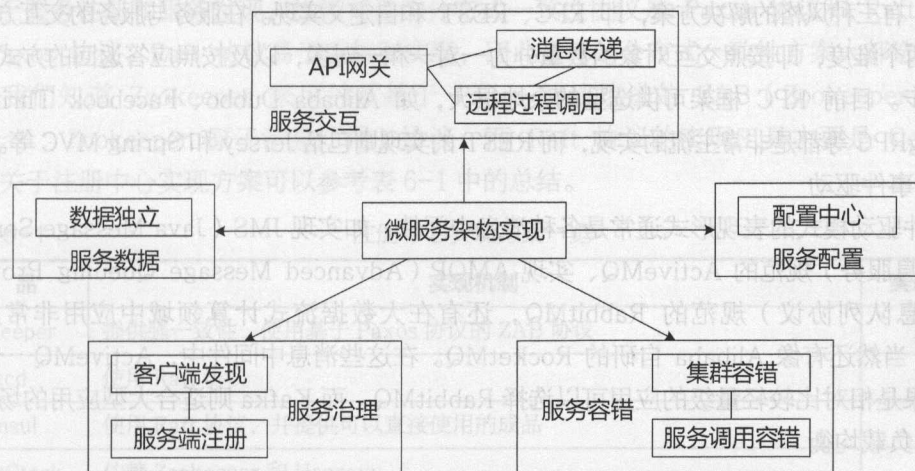


图 6-1 微服务架构实现需求

图 6-1 中，我们可以看到服务注册和发现、服务间通信、服务容错、数据管理、API 网关等构成了一个完整微服务架构需要实现的核心功能。本章首先将对微服务架构实现技术进行选型，然后基于选定工具和框架对这些功能的实现做具体展开。

## 6.1 微服务架构实现技术选型

实现微服务架构的第一步是技术选型，也就是选择一个合适的技术体系来支持微服务的开发工作。目前市面上并没有一个真正意义上实现微服务的技术体系，但还是存在一些可供参考的工具和框架。本书后续内容将使用 Spring Cloud 作为实现微服务的主体框架，而在本节我们会给出为什么选择该框架的原因，同时也会对其他同类框架做分析和类比。





## 6.1.1 技术选型的参考标准

所谓技术选型，首先需要明确选型的参考标准。结合我们在第 4 章至第 5 章对微服务本身的讨论，我们对于工具的选择也将围绕核心组件完备性和关键要素实现难度两方面来展开。

### 1. 核心组件完备性

微服务的核心组件由服务通信、事件驱动、负载均衡、服务路由、API 网关和配置管理所组成。

#### • 服务通信

对于服务通信，微服务架构明确要求服务之间通过跨进程的远程调用方式进行通信。关于远程调用，有三种风格的解决方案，即 RPC、REST 和自定义实现。在服务与服务的交互方式上，也存在两个维度，即按照交互对象的数量分为一对一和一对多，以及按照应答返回的方式分为同步和异步。目前 RPC 框架可供选型的余地很大，如 Alibaba Dubbo、Facebook Thrift 以及 Google gRPC 等都是非常主流的实现，而 REST 的实现则包括 Jersey 和 Spring MVC 等。

#### • 事件驱动

事件驱动模式的表现形式通常是各种消息中间件，如实现 JMS（Java Message Service，Java 消息服务）规范的 ActiveMQ、实现 AMQP（Advanced Message Queuing Protocol，高级消息队列协议）规范的 RabbitMQ，还有在大数据流式计算领域中应用非常广泛的 Kafka，当然还有像 Alibaba 自研的 RocketMQ。在这些消息中间件中，ActiveMQ 一般不考虑，如果是相对比较轻量级的应用可以选择 RabbitMQ，而 Kafka 则适合大型应用的场景。

#### • 负载均衡

负载均衡分为服务器端负载均衡和客户端负载均衡两大类实现方案。从服务器软件上，我们可以选择 Nginx、HA Proxy、Apache、LVS 等工具。而 Netflix Ribbon 则是一种单独可以使用的客户端负载均衡机制。事实上所有的分布式服务框架基本都内置了负载均衡的实现，所以负载均衡本身并不需要做太多的选择。

#### • API 网关

API 网关是微服务架构的核心组件之一。Spring 旗下有一个 Spring Cloud Netflix Zuul 提供了一套过滤器机制，可以很好地支持签名校验、登录校验等前置过滤功能处理，同时它也维护了路由规则和服务实例，以便完成服务路由功能。其他可供参考的 API 网关还有 Mashape 和开源的 Kong 等。

#### • 配置管理

配置管理的作用是完成集中式的配置信息管理。目前比较主流的包括 Spring 旗下的 Spring Cloud Config、淘宝的 Diamond 和百度的 Disconf 等。在实现上，Spring Cloud



Config 支持配置信息放在配置服务的内存中（即本地），也支持放在远程 Git 仓库，这点与其他工具在设计上有较大不同。Diamond 和 Disconf 都是基于 Mysql 作为存储媒介，Diamond 采用拉模型，即每隔 15s 拉一次全量数据；而 Disconf 基于 Zookeeper 的推模型，实时推送。在配置数据模型上，Diamond 只支持 Key-Value 结构的数据，非配置文件模式；Disconf 支持传统的配置文件模式，也支持 Key-Value 结构数据。

## 2. 关键要素实现难度

关于微服务实现的关键要素，因为数据一致性更多是一种实现机制而不是具体工具，所以我们主要需要明确注册中心以及服务可靠性实现上的难度。

### • 注册中心

关于服务注册和服务发现，比较常见的分布式一致性协议是 Paxos 协议和 Raft 协议。相比 Paxos 协议，Raft 协议易于理解和实现，因此最新的分布式一致性方案大都选择 Raft 协议。我们知道 Zookeeper 采用的是基于 Paxos 协议改进的 ZAB（Zookeeper Atomic Broadcast，Zookeeper 原子消息广播）协议，而 Raft 协议的实现工具主要是 Consul 和 Etcd。关于注册中心实现方案可以参考表 6-1 中的总结。

表 6-1 注册中心实现方案一览

产 品	实现机制	实现语言
Zookeeper	提供强一致性，使用基于 Paxos 协议的 ZAB 协议	Java
Etcd	使用 Raft 协议	Go
Consul	使用 Raft 协议，并提供可以直接使用的成品	Go
SmartStack	依赖 Zookeeper 和 Haproxy	Go
Eureka	来自 Netflix，采用自身的一套实现机制，且客户端和服务端都是基于 Java 实现，只能用于基于 JVM 的开发环境	Java
Serf	采用基于 Gossip 的 SWIM 协议	Go

注册中心是任何一个微服务框架必不可少的组件，很多框架都内建了对服务注册中心的支持。例如，Alibaba 的 Dubbo 框架支持包括 Zookeeper、Redis 等在内的多种注册中心实现，默认采用的是 Zookeeper；新浪的 Motan 支持 Zookeeper，也支持 Consul；Spring Cloud 同时提供了 Spring Cloud Consul 和 Spring Cloud Zookeeper 两种实现方案；Netflix OSS 中使用的是 Eureka。

### • 服务可靠性

服务可靠性相关的功能主要包括服务容错、服务隔离、服务限流和服务降级，其中大多数机制都偏向于实现策略而不是实现工具。我们需要明确的是如何实现服务隔离和服务熔断机制



的框架。

服务熔断器目前只有一个可选的开源方案，那就是 Hystrix。Hystrix 同时也实现了服务隔离机制。本章后续会对该框架做详细分析。

接下去我们就通过以上的选型标准以及一些主流工具的对比来梳理主流的微服务实现框架。

## 6.1.2 微服务实现框架对比

讲完微服务框架应该具备的核心组件和关键要素之后，我们来看一下目前业界有哪些现成的框架可供我们直接使用，这里列举了 Dubbo 和 Spring Cloud 这两个目前最主流框架作为我们选型的基础。

### 1. Dubbo

Dubbo 是国内 SOA 框架集大成之作，基本具备一个 SOA 框架应有的所有功能，包括高性能通信、多协议集成、服务注册与发现、服务路由、负载均衡、服务治理等核心功能。作为一个 RPC 架构和 SOA 架构，Dubbo 无疑是非常优秀的，但在功能完备性上，API 网关、服务熔断器等核心组件在 Dubbo 中并没有完整体现。

从社区活跃度上，Dubbo 在 2012 年年底已经基本不再更新，但不影响其在各大互联网公司的应用和扩展。好消息是最近 Alibaba 宣布重新启动 Dubbo 的维护工作。

Dubbo 的文档可以说在国内开源框架中算是一流的，非常全面且讲解得比较深入，由于版本已经稳定不再更新，所以也不太会出现不一致的情况，学习成本较低。

### 2. Spring Cloud

Spring Cloud 是 Spring 家族中新的一员，重点打造面向服务化的功能组件，在功能上服务注册中心、API 网关、服务熔断器、分布式配置中心等组件都能在 Spring Cloud 中找到对应的实现。

从版本更新上，显然 Spring Cloud 也表现得非常活跃。目前 Spring Cloud 在 Github 的托管代码几乎每天都有更新，其发展仍处于高速迭代的阶段。

Spring Cloud 在一定程度上是一种集成型的框架，其内部大量依赖了各种外部的第三方工具和框架，所以文档在体量上自然要比 Dubbo 多很多，文档内容上还算简洁清楚，更多的是偏向整合，更深入的使用方法还是需要查看第三方组件的详细文档。

### 3. 对比与结论

通过对比分析，本书将选择 Spring Cloud 作为我们实现微服务架构的主体框架。功能的完备性是我们选择 Spring Cloud 的主要原因，表 6-2 中列出了 Dubbo 与 Spring Cloud 功能



对比一览，可以看到 Spring Cloud 提供了很多 Dubbo 所不具备但对微服务实现又必不可少的核心组件。

表 6-2 Dubbo 与 Spring Cloud 功能一览

	Dubbo	Spring Cloud
服务注册中心	Zookeeper	Spring Cloud Netflix Eureka
服务调用方式	RPC	RESTful API
API 网关	无	Spring Cloud Netflix Zuul
服务熔断器	不完善	Spring Cloud Netflix Hystrix
分布式配置	无	Spring Cloud Config
服务跟踪	无	Spring Cloud Sleuth
消息总线	无	Spring Cloud Bus
数据流	无	Spring Cloud Stream
安全性	无	Spring Cloud Security
...	...	...

另一个选择 Spring Cloud 的原因在于服务之间的交互方式。我们知道微服务架构中推崇基于 HTTP 协议的 RESTful 风格实现服务间通信，而 Dubbo 的服务调用通过 RPC 实现。采用 RPC 方式会导致服务提供方与调用方接口产生较强依赖，而且服务对技术敏感，无法做到非常通用。Spring Cloud 采用的就是 RESTful 风格，这方面更加符合微服务架构的设计理念。

Spring Cloud 还具备一个天生的优势，因为它是 Spring 家庭的一员，而 Spring 在开发领域的强大地位给 Spring Cloud 起到很好的推动作用。同时，Spring Cloud 基于 Spring Boot，而 Spring Boot 目前已经在越来越多的公司得到应用和推广，用来简化 Spring 应用的框架搭建以及开发过程。Spring Cloud 也继承了 Spring Boot 简单配置、快速开发、轻松部署的特点，让原本复杂的架构工作变得相对容易上手。

通过上面对几个环节的分析，相信读者对 Dubbo 和 Spring Cloud 有了一个初步的了解。从目前 Spring Cloud 的被关注度和活跃度上来看，其很有可能将来会成为微服务架构的标准框架。本章后续内容将对 Spring Cloud 的部分核心组件做详细展开，在此之前我们先来了解一下 Spring Boot。

## 6.2 Spring Boot

在本节中，我们将推荐 Spring Boot 作为实现微服务架构的基础框架。目前，Spring



Boot 被越来越多的开发团队所采用，用于替代原有的 Spring 框架。而在微服务架构中，Spring Boot 也是构成 Spring Cloud 的基础。

## 6.2.1 Spring Boot 概览

在引入 Spring Boot 之前，我们先来回顾一下使用 Spring 开发 Web 应用程序的过程，通常包括使用 Maven、Gradle 等工具搭建工程、web.xml 定义 Spring 的 DispatcherServlet、完成启动 Spring MVC 的配置文件、编写响应 HTTP 请求的 Controller 以及将服务部署到 Tomcat Web 服务器等步骤。Spring 框架诞生以来，以上开发过程被广泛采用并形成了一套固定的开发模式。但开发过程同样在不断发展，基于传统的 Spring 框架进行开发，逐渐暴露出一些问题，比较典型的就是过于复杂和繁重的配置工作。

如果想要优化这一套开发过程，有几个点值得我们去挖掘，如使用约定优于配置（Convention Over Configuration）思想的自动化配置、启动依赖项自动管理、简化部署并提供应用监控等。这些优化点推动了以 Spring Boot 为代表的新一代开发框架的诞生，作为 Spring 家族新的一员，Spring Boot 提供了令人兴奋的特性，这些特性主要体现在开发过程的简单之美。例如，支持快速构建项目、不依赖外部容器独立运行、开发部署效率高以及与云平台天然集成，这些特点促使我们选择 Spring Boot 来构建微服务。Spring Boot 的核心优势体现在编码、配置、部署、监控等多个方面。

Spring Boot 使编码更简单，我们只需要在 Maven 中添加一项依赖并实现一个方法就可以提供微服务架构中所推崇的 RESTful 风格接口。

Spring Boot 使配置更简单，把 Spring 中基于 XML 的功能配置方式转换为 Java Config，把基于 \*.properties/\*.xml 文件的部署环境配置转换成语义更为强大的 \*.yaml。同时，对常见的各种功能组件均提供了各种默认的 starter 依赖以简化 Maven 配置。

Spring Boot 使部署更简单。Spring Boot 部署包结构参考图 6-2，我们可以看到相较于传统模式下的 war 包，Spring Boot 部署包既包含了业务代码和各种第三方类库，同时也内嵌了 HTTP 容器。这种包结构支持 java-jar standalone.jar 方式的一键启动，不需要预部署应用服务器，通过默认内嵌 Tomcat 降低对运行环境的基本要求。

Spring Boot 使监控更简单，基于 spring-boot-actuator 组件，可以通过 RESTful 接口以及 HATEOAS 表现方式获取 JVM 性能指标、线程工作状态等运行时信息。例如，可以通过 HTTP GET /env/{name} 接口获取系统环境变量、通过 HTTP GET

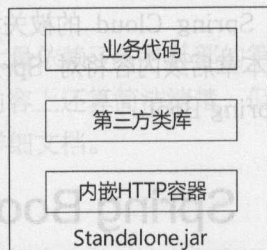


图 6-2 Spring Boot 部署包结构

仅供对书籍质量进行鉴别！是否为购买正版实体书提供依据！！  
非卖品！！严禁（售卖和上传互联网平台）！！



/mapping 接口获取所有 RESTful 服务、通过 HTTP GET /dump 获取线程工作状态以及通过 HTTP GET /metrics/{name} 获取 JVM 性能指标等。

## 6.2.2 Spring Boot 核心原理

通过 @SpringBootApplication 注解并配合各种内置的代码工程，我们可以利用 Spring Boot 快速打造一个复杂系统构建过程中所需的各种功能体系。本节中，我们将分析 @SpringBootApplication 注解背后的实现原理。

基于 Spring Boot 的开发模式非常简单。例如，通过引入 spring-boot-starter-web 工程，并在应用程序入口添加 @SpringBootApplication 注解，就能构建 RESTful 风格接口，这在 Spring 框架中需要通过各种配置和代码才能实现同样的功能。

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

```
@Controller
@SpringBootApplication
public class Application {
    public static void main(String[] args) {
        SpringApplication.run(Application.class, args);
    }

    @RequestMapping("/")
    String home() {
        return "Hello Spring Boot!";
    }
}
```

显然，上述代码能够运行的关键在于一个强大的注解 @SpringBootApplication，@SpringBootApplication 注解的定义如下。我们可以看到该注解实际上是由三个注解组合而成，分别是 @Configuration、@EnableAutoConfiguration 和 @ComponentScan。

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.RUNTIME)
@Documented
@Inherited
@Configuration
@EnableAutoConfiguration
@ComponentScan
public @interface SpringBootApplication
```

在 Spring 中，@Configuration 注解比较常见，提供 JavaConfig 配置类实现。而 @ComponentScan 则扫描 @Component 等注解，把相关 bean 定义批量加载到 IoC 容器中。这里重点需要剖析的是 @EnableAutoConfiguration 注解，该注解的定义如下，我们看到该注解中通过



@Import 注解引入了 EnableAutoConfigurationImportSelector 类。@Import 注解在 Spring 中用于导入 JavaConfig 配置类，而在这里导入的就是 EnableAutoConfigurationImportSelector 类。

```
@Target(value=TYPE)
@Retention(value=RUNTIME)
@Documented
@Inherited
@AutoConfigurationPackage
@Import(value=EnableAutoConfigurationImportSelector.class)
public @interface EnableAutoConfiguration
```

从类名上看，EnableAutoConfigurationImportSelector 类是一种选择器，负责从各种配置项中找到需要导入的具体配置类。EnableAutoConfigurationImportSelector 类的结构如图 6-3 所示，其所依赖的最关键组件是 SpringFactoriesLoader。

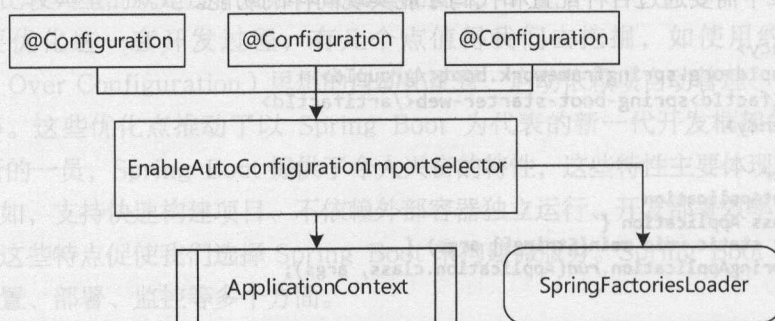


图 6-3 EnableAutoConfigurationImportSelector 结构

要想理解 SpringFactoriesLoader 类，首先需要了解 JDK 中 SPI (Service Provider Interface，服务提供者接口) 机制。JDK 提供了服务实现查找的一个工具类 java.util.ServiceLoader 来实现 SPI 机制。我们可以在服务的提供者提供了服务接口的一种实现之后，在 jar 包的 META-INF/services/ 目录同时创建一个以服务接口命名的文件，该文件里配置着一组 Key-Value，用于指定服务接口与实现该服务接口具体实现类的映射关系。而当外部程序装配这个模块的时候，就能通过该 jar 包 META-INF/services/ 里的配置文件找到具体的实现类名，并装载实例化，完成模块的注入。基于这样一个约定，就能很好地找到服务接口的实现类，而不需要在代码里硬编码指定。SpringFactoriesLoader 类似这种 SPI 机制，只不过服务接口命名的文件放在 META-INF/spring.factories 文件夹下，对应的 Key 为 EnableAutoConfiguration。因此，SpringFactoriesLoader 会查找所有 META-INF/spring.factories 配置文件，并把 Key 为 EnableAutoConfiguration 所对应的配置项通过反射实例化为配置类并加载到容器。以下就是 Spring Boot 中所使用的 spring.factories 配置文件片段，可以看到 EnableAutoConfiguration 中包括了各式各样的配置项，这些配置项在



Spring Boot 启动过程中都能够通过 SpringFactoriesLoader 加载到运行时环境，从而实现自动化配置。

```
# Auto Configure
org.springframework.boot.autoconfigure.EnableAutoConfiguration=\
org.springframework.boot.autoconfigure.aop.AopAutoConfiguration,\
org.springframework.boot.autoconfigure.amqp.RabbitAutoConfiguration,\
org.springframework.boot.autoconfigure.MessageSourceAutoConfiguration,\
org.springframework.boot.autoconfigure.PropertyPlaceholderAutoConfiguration,\
org.springframework.boot.autoconfigure.batch.BatchAutoConfiguration,\
org.springframework.boot.autoconfigure.data.jpa.JpaRepositoriesAutoConfiguration,\
org.springframework.boot.autoconfigure.data.mongo.MongoRepositoriesAutoConfiguration,\
org.springframework.boot.autoconfigure.redis.RedisAutoConfiguration,\
```

以上就是 Spring Boot 实现自动配置的基本过程和原理。当然，@SpringBootApplication 注解也可以基于外部配置文件加载配置信息。基于约定优于配置思想，Spring Boot 在加载外部配置文件的过程中大量使用了默认配置。

针对一般的服务化开发模式，Spring Boot 为我们提供了数据访问、消息传递以及其内置的监控系统等基本功能。使用 Spring Boot 作为构建微服务的轻量级框架是合适的，但 Spring Boot 并不包含服务注册和发现机制、外围服务监控、服务熔断和服务治理等功能，提供这些功能的是 Spring Cloud。

## 6.3 Spring Cloud

Spring Cloud 基于 Spring Boot，作为本书对微服务架构进行技术选型的结果，我们将通过较大的篇幅介绍 Spring Cloud 框架。

### 6.3.1 Spring Cloud 概览

Spring Cloud 是一系列框架的有序集合。它利用 Spring Boot 的开发便利性巧妙地简化了分布式系统基础设施的开发过程，如服务发现注册、配置中心、消息总线、负载均衡、断路器、数据监控等都可以使用 Spring Boot 的开发风格做到一键启动和部署。

在对 Spring Cloud 框架进行设计和实现的过程中，Spring 并没有重复制造轮子，它只是将目前各家公司开发得比较成熟、经得起实际考验的服务框架组合起来，通过 Spring Boot 风格进行再封装，从而屏蔽掉了复杂的配置和实现原理，最终给开发者留出了一套简单易懂、易部署和易维护的分布式系统开发工具包。

对于那些没有实力或者没有足够的资金投入去开发自己的分布式系统基础设施的公司而言，使用 Spring Cloud 一站式解决方案能在从容应对业务发展的同时大大减少开发成本。



Spring Cloud 标准化的、全站式的技术方案构成了一个生态圈，涵盖众多微服务架构实现所需的核心组件。

- Spring Cloud Config

配置管理开发工具包，可以把配置信息放到远程服务器，目前支持本地存储、Git 以及 Subversion。

- Spring Cloud Bus

事件消息总线，用于在集群中传播状态变化（如配置信息变更事件），可与 Spring Cloud Config 联合实现热部署。

- Spring Cloud Netflix

针对多种 Netflix 组件提供的开发工具包，其中包括 Eureka、Hystrix、Zuul 等。

- Netflix Eureka

一个基于 REST 的服务注册中心，用于定位服务以及提供中间层服务器的故障转移。

- Netflix Hystrix

容错管理工具，旨在通过控制服务之间的依赖关系，从而对延迟和故障提供更强大的容错能力。

- Netflix Zuul

API 网关工具，提供动态路由、监控、弹性、安全等边缘服务。

- Netflix Archaius

配置管理 API，包含一系列配置管理 API，提供动态类型化属性、线程安全配置操作、轮询框架、回调机制等功能。

- Spring Cloud for Cloud Foundry

通过 OAuth2 协议绑定服务到 Cloud Foundry，Cloud Foundry 是 VMware 推出的开源 PaaS 云平台。

- Spring Cloud Sleuth

日志收集工具包，封装了 Dapper、Zipkin 和 HTrace 操作。

- Spring Cloud Data Flow

大数据操作工具，通过命令行方式操作数据流。

- Spring Cloud Security

安全工具包，为应用程序添加安全控制，主要提供 OAuth2 协议的支持。

- Spring Cloud Consul

封装了 Consul 操作，Consul 是一个服务发现与配置工具，与 Docker 容器可以无缝集成。

- Spring Cloud Zookeeper

操作 Zookeeper 的工具包，用于基于 Zookeeper 方式的服务注册和发现。



- Spring Cloud Stream

数据流操作开发包,封装了如 Redis、RabbitMQ、Kafka 等工具以支持发送接收消息。

- Spring Cloud CLI

基于 Spring Boot CLI,支持以命令行方式快速建立云组件。

我们不会对以上所有的组件进行展开,本节介绍 Spring Cloud 的基本思路还是围绕着第 4 章和第 5 章中微服务的核心组件和关键要素对相关工具包的实现方法和基本原理进行分析,涉及 Spring Cloud Netflix、Spring Cloud Config、Spring Cloud Sleuth 等组件,其中 Spring Cloud Netflix 又是对 Netflix 旗下的 Eureka、Ribbon、Hystrix、Feign 和 Zuul 等组件的封装,我们会从这些原始组件的基本原理出发,结合 Spring Cloud Netflix,对它们的封装内容进行介绍(见图 6-4)。

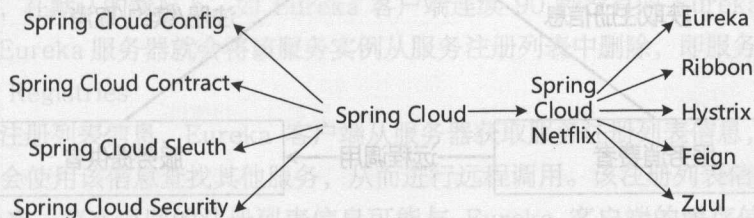


图 6-4 本书中的 Spring Cloud 组件

另外,图 6-4 中与微服务跟踪相关的 Spring Cloud Sleuth、微服务安全性相关的 Spring Cloud Security 以及面向消费者驱动的契约测试组件 Spring Cloud Contract 等偏向管理体系方面的部分组件将放到下一章中统一介绍。

### 6.3.2 Spring Cloud Netflix Eureka 与服务治理

Spring Cloud Netflix Eureka 基于 Netflix Eureka。我们已经在 6.1.1 节关于微服务架构实现技术选型中提到了 Netflix Eureka,采用自身的一套实现机制,且客户端和服务端都是采用 Java 实现,一般只能用于基于 JVM 的开发环境。当然,由于 Eureka 服务器端同样提供了完备的 RESTful API,所以也支持将非 Java 语言构建的微服务纳入服务治理体系中来。本节关于 Spring Cloud Netflix Eureka 与服务治理的讨论主要围绕 Netflix Eureka 展开。

#### 1. Netflix Eureka 基本架构

Eureka 是由 Netflix 开源的一款提供服务注册和发现的产品,其基本架构如图 6-5 所示。我们可以看到该架构由 3 个逻辑角色组成,并与 5.1.1 节中介绍的服务注册中心模型具有



高度一致性。

- Eureka Server

提供服务注册和发现。

- Service Provider

服务提供方，将自身服务注册到 Eureka，从而使服务消费方能够找到。

- Service Consumer

服务消费方，从 Eureka 获取注册服务列表，从而实现消费服务。

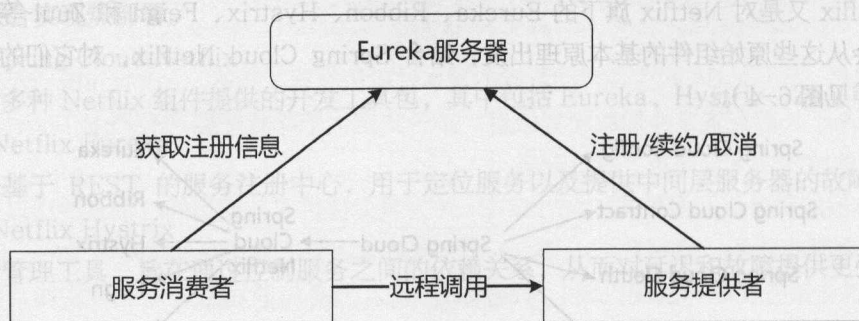


图 6-5 Netflix Eureka 基本架构

我们再对图 6-5 中的基本架构做进一步展开，可以得到如图 6-6 所示的细化架构，该图进一步展示了 3 个角色之间的交互：首先 Service Provider 会向 Eureka Server 提交服务管理相关操作；而 Eureka Server 之间会做注册服务的同步，从而保证状态一致；同时，Service Consumer 会向 Eureka Server 获取注册服务列表，并消费服务。

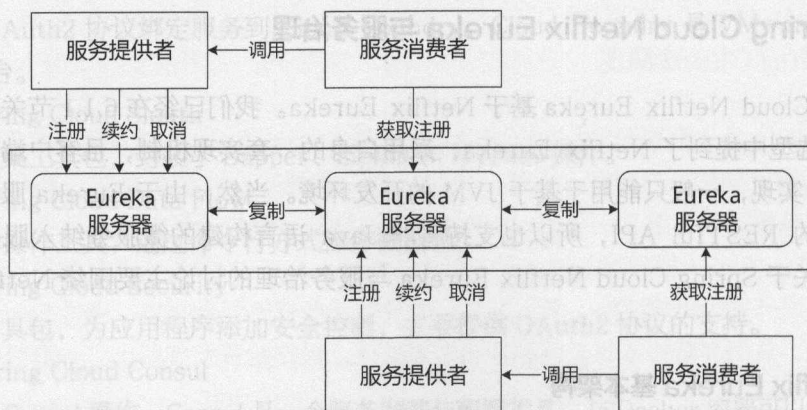


图 6-6 Netflix Eureka 细化架构

Eureka 有以下几个概念与服务治理直接相关。



- Register

服务注册, 当 Eureka 客户端向 Eureka Server 注册时, 它提供自身的元数据, 如 IP 地址、端口、运行状况指示符 URL 等信息。

- Renew

服务续约, Eureka 客户端会每隔 30 秒发送一次心跳来进行服务续约。通过续约来告知 Eureka Server 该客户端仍然存在, 希望服务器不要剔除自己。

- Cancel

服务下线, Eureka 客户端在程序关闭时向 Eureka 服务器发送取消请求。发送请求后, 该客户端实例信息将从服务器的实例注册列表中删除。

- Eviction

服务剔除, 在默认的情况下, 如 Eureka 客户端连续 90 秒没有向 Eureka 服务器发送服务续约心跳, Eureka 服务器就会将该服务实例从服务注册列表中删除, 即服务剔除。

- Fetch Registries

获取服务注册列表信息, Eureka 客户端从服务器获取服务注册列表信息, 并将其缓存在本地。客户端会使用该信息查找其他服务, 从而进行远程调用。该注册列表信息定期(每 30 秒钟)更新一次。每次返回的注册列表信息可能与 Eureka 客户端的缓存信息有所不同, Eureka 客户端会自动处理两者之间的差异。

显然, 对于一个注册中心而言, 想要完整理解全局性的设计理念和原理, 我们需要分别从注册中心服务器、服务提供者、服务消费者这三个角色出发, 分析它们之间的交互细节。

### (1) 注册中心服务器

想要实现服务管理, 首先必须包含服务的定义, 在 Eureka 中使用两层 HashMap 来达到服务存储的效果。第一层 HashMap 的 Key 是 App Name, 也就是应用名字, 而第二层 HashMap 的 Key 是 Instance Name, 也就是实例名字。

面向服务注册中心的操作主要包括服务注册、服务续约和服务下线, 实际上这三种不同的操作对于注册中心服务器而言其执行的工作流程基本一致, 它们都是对服务存储的操作, 并把这一操作同步到其他 Eureka 节点。以服务注册为例, 图 6-7 展示了主要几个组件之间的交互过程以及核心操作。服务续约是一个定时操作, 类似于心跳(Heartbeat)。Eureka 会定期(默认为每 60 秒)检测失效的服务, 检测标准就是超过一定时间没有续约的服务。而服务下线用来把自身的服务从 Eureka Server 中删除, 以防客户端调用不存在的服务。



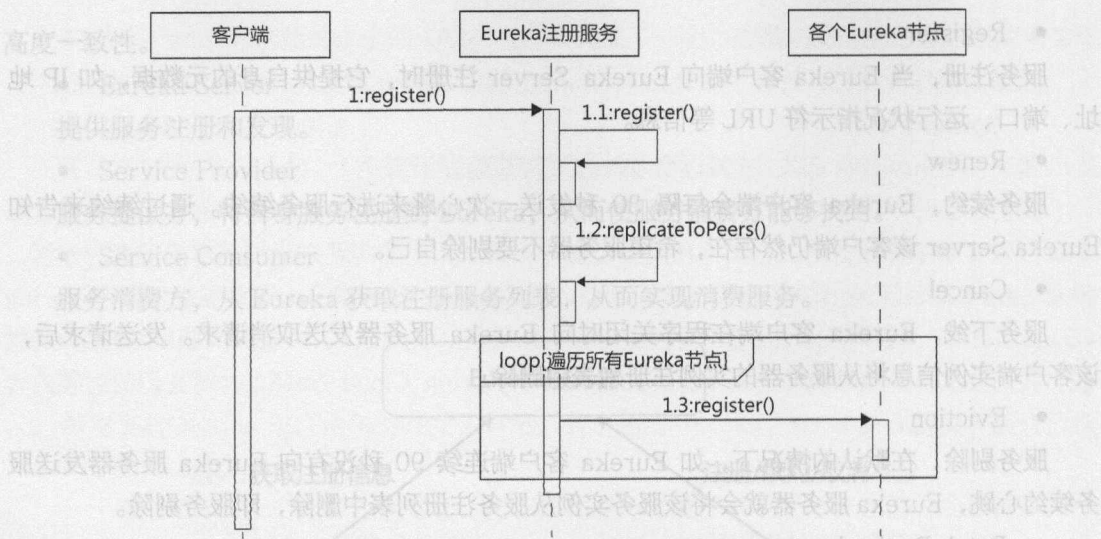


图 6-7 Netflix Eureka 服务注册时序图

注册中心服务器的另一个核心功能是提供服务列表，为了提高性能，服务列表在 Eureka Server 会缓存一份，同时每 30 秒更新一次。

### (2) 服务提供者

服务提供者关注服务注册、服务续约和服务下线等功能，相对注册中心服务器，服务提供者相当于是客户端，所以它可以使用服务器提供的 RESTful API（基于 Jersey 框架实现）完成上述操作。我们以基于心跳操作的服务续约为例给出服务提供者的大致交互流程，见图 6-8。

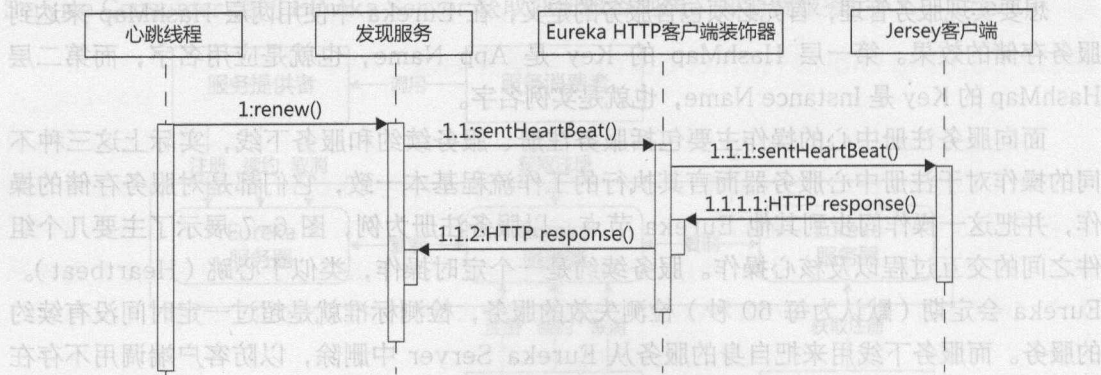


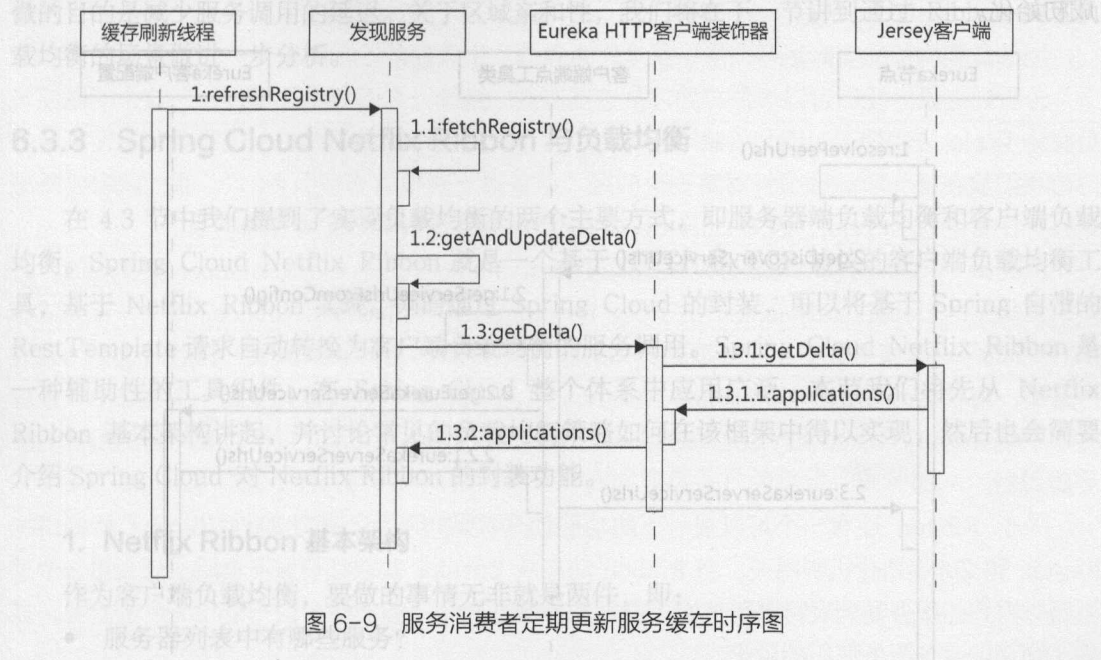
图 6-8 服务提供者服务续约时序图

### (3) 服务消费者

服务消费者与服务提供者本质上是同一个客户端。如同 5.1.1 节中介绍的注册中心实现方



案, 服务消费者在本地会有一份缓存, 所以需要定期更新缓存。更新操作的流程图见图 6-9, 可以看到它与图 5-1 非常一致。



## 2. Netflix Eureka 高可用

在微服务架构中, 我们需要考虑发生故障的情况, 所以在生产环境中我们需要对服务中各个组件进行高可用部署。服务注册中心虽然是一个侧重于服务治理的辅助性组件, 同样也需要确保可用性。

Eureka Server 的高可用实际上就是将自己作为服务向其他服务注册中心注册自己, 这样就形成了一组互相注册的服务中心, 以实现服务列表的互相同步, 达到高可用的效果。

我们在图 6-7 中看到一个非常重要的 `replicateToPeers` 操作, 该操作用来实现服务器节点之间的状态同步。通过这种方式, 任意一个 Eureka Server 获取来自客户端的通知之后就能保证状态会在所有的 Eureka Server 中得到更新。具体实现方式其实很简单, 就是接收到请求的 Eureka Server 把请求再次转发到其他的 Eureka Server, 调用同样的接口, 传入同样的参数, 唯一不同之处在于会告诉其他服务器不需要再进行复制。

剩下的一个问题就是我们如何知道有哪些 Eureka Server 节点。Eureka Server 在启动后会调用 `EurekaClientConfig.getEurekaServerServiceUrls` 来获取所有的 Peer 节点, 并且会定期更新 (见图 6-10)。这个方法的默认实现是从配置文件读取, 所以如果 Eureka Server 节点相对固定, 可以通过在配置文件中配置节点列表的方式来实现。如果在运行过程中新加入



了一个节点，或者服务器重启，那么每个节点可以把自己当作是服务消费者从其他 Eureka Server 节点获取所有服务的注册信息。然后再对每个服务在自己这里执行注册操作，从而完成初始化。

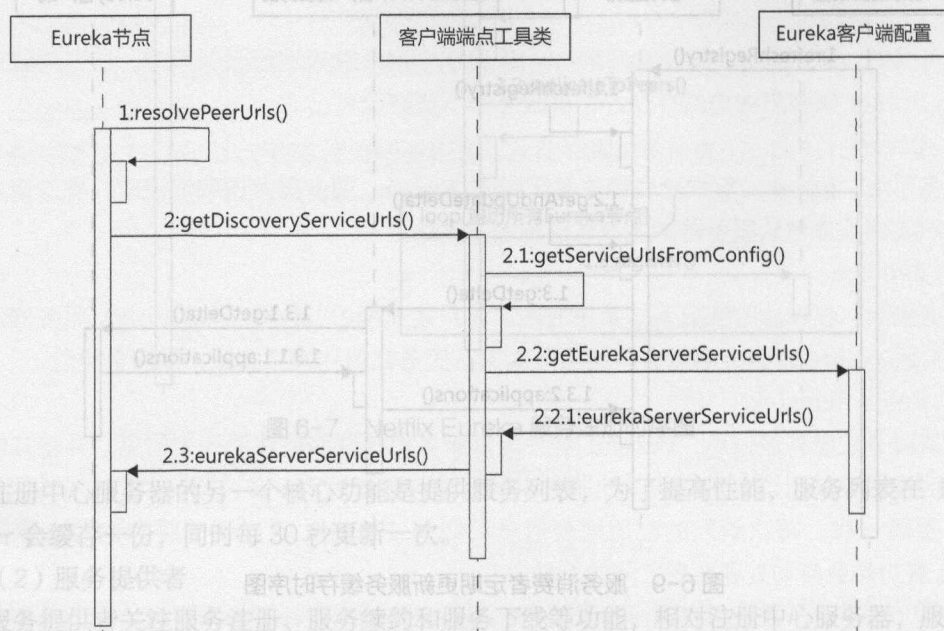


图 6-10 Netflix Eureka 获取服务器节点时序图

### 3. Netflix Eureka 区域亲和性

在 Eureka 存在 Region 和 Zone 的概念，我们可以简单地将 Region 理解为 Eureka 集群，Zone 理解成机房。结合图 6-11 中的结构图，可以看到一个 Eureka 集群被部署在了 Zone1 和 Zone2 两个机房中。

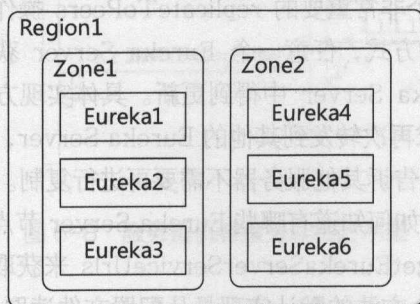


图 6-11 Netflix Eureka 中的 Region 和 Zone



由 Region 和 Zone 的关系我们可以得出区域亲和性（Zone Affinity）的概念，所谓区域亲和性，就是指服务消费者在选择服务时会选择和该服务消费者在同一个 Zone 的服务，这样做的目的是减少服务调用的延迟。关于区域亲和性，我们将在下一节讲到通过 Ribbon 实现负载均衡的场景做进一步分析。

### 6.3.3 Spring Cloud Netflix Ribbon 与负载均衡

在 4.3 节中我们提到了实现负载均衡的两个主要方式，即服务器端负载均衡和客户端负载均衡。Spring Cloud Netflix Ribbon 就是一个基于 HTTP 和 TCP 协议的客户端负载均衡工具，基于 Netflix Ribbon 实现。同时通过 Spring Cloud 的封装，可以将基于 Spring 自带的 RestTemplate 请求自动转换为客户端负载均衡的服务调用。Spring Cloud Netflix Ribbon 是一种辅助性的工具组件，在 Spring Cloud 整个体系中应用广泛。本节我们将先从 Netflix Ribbon 基本架构讲起，并讨论常见的负载均衡策略如何在该框架中得以实现，然后也会简要介绍 Spring Cloud 对 Netflix Ribbon 的封装功能。

#### 1. Netflix Ribbon 基本架构

作为客户端负载均衡，要做的事情无非就是两件，即：

- 服务器列表中有哪些服务？
- 如何从这些服务中选择一个服务进行调用？

Netflix Ribbon 的核心接口 `ILoadBalancer` 就是围绕这两个问题来设计的，包括了如下基本方法。

- `addServers`

配置后端服务。

- `chooseServer`

选择一个后端服务。

- `markServerDown`

标记一个服务不可用。

- `getServerList`

获取后端服务列表。

`ILoadBalancer` 接口最基本的实现类是 `BaseLoadBalancer`，可以说负载均衡的核心功能都在这个类中得以实现，而该类同样包含了作为一个负载均衡器应该具备的一些核心组件。

- `IRule`

负载均衡策略，可以插件化地为负载均衡提供各种适用的负载均衡算法。



- IPing

判断目标服务是否存活。

- LoadBalancerStats

记录负载均衡的实时运行信息，用来作为负载均衡策略的运行输入。

分析到这里，我们可以想象“如何从这些服务中选择一个服务进行调用？”这个问题应该是交给 IRule 去实现了，因为负载均衡算法决定了如何选择服务的结果，关于负载均衡策略的讨论我们放在下一节。那么针对第一个问题“服务器列表中有哪些服务？”，我们就需要联想到服务注册中心 Eureka，因为 Eureka 中存放着系统当前所有的服务注册信息。

Ribbon 中动态加载后端服务列表是通过 DynamicServerListLoadBalancer 类实现的，它是前面提到的 BaseLoadBalancer 的子类。DynamicServerListLoadBalancer 类中，首先依靠 EurekaClient 从服务注册中心获取到具体的服务实例 InstanceInfo 列表，然后对这个列表进行遍历，将状态为在线的实例转换成 DiscoveryEnabledServer 对象并放到一个集合中，最后将这个集合返回。DiscoveryEnabledServer 对象的集合就是 Ribbon 进行负载均衡的所需的服务器列表。

关于 Ribbon 还有一个可以提一下的功能就是区域亲和性，该功能与 Eureka 所提供的 Region 和 Zone 划分机制相关。当 Ribbon 动态从 Eureka 获取注册服务器后，还会用一个过滤器对得到的服务器列表再进行过滤，根据配置会选择和服务消费者在一个 Zone 的服务，显然这样可以减少服务调用的延迟。

## 2. Netflix Ribbon 负载均衡策略

当获取到服务器列表之后，我们就可以采用一定的负载均衡策略解决“如何从这些服务中选择一个服务进行调用？”。Ribbon 把这个工作委托给了 IRule 接口的实现类，不同的 IRule 实现类可以提供不同的负载均衡算法。

Ribbon 提供了常用的几种负载均衡策略，包括 RoundRobinRule、RandomRule 这样不依赖于服务器运行状况的静态策略，也有 AvailabilityFilteringRule、WeightedResponseTimeRule 等多种基于收集到的服务器运行状况进行决策的动态策略。表 6-3 对这些负载均衡策略做了总结 and 对比。

表 6-3

Ribbon 自带负载均衡策略一览

策略名	策略描述	实现说明
BestAvailableRule	选择一个最小的并发请求的服务器	逐个考察服务器，再选择其中活跃请求数最小的服务器
AvailabilityFilteringRule	过滤掉那些一直处于连接失败和高并发状态的后端服务器	检查 LoadBalancerStats 里记录的各个服务器的运行状态



续表

策略名	策略描述	实现说明
WeightedResponseTimeRule	根据响应时间分配一个权重，响应时间越长，权重越小，被选中的可能性越低	一个后台线程定期地从 LoadBalancerStats 里面读取平均响应时间，为每个服务器计算一个权重
RetryRule	对选定的负载均衡策略执行重试机制	在一个可配置的时间段内，每当选择服务器不成功，则一直尝试选择一个可用的服务器
RoundRobinRule	轮询方式选择服务器	轮询服务器列表中的位置下标，选择下标对应位置的服务器
RandomRule	随机选择一个服务器	随机选择下标，选择下标对应位置的服务器

3. Spring Cloud Netflix Ribbon 功能

基于 Spring Cloud Netflix Ribbon，有两种方式实现客户端负载均衡功能，一种是使用 @RibbonClient 注解命名化客户端，代码示例如下。

```
@Configuration
@RibbonClient(name = "foo", configuration = FooConfiguration.class)
public class TestConfiguration {
    // ...
}
```

另一种是注入 RestTemplate 时自动使用 Ribbon，代码示例如下。

```
@Autowired
RestTemplate restTemplate;

public Product getProduct(String productName) {
    return restTemplate.getForObject("http://product/" + productName, Product.class);
}
```

因为 Netflix Ribbon 本质上只是一个工具，而不是一套完整的解决方案，所以 Spring Cloud Netflix Ribbon 对 Netflix Ribbon 做了封装。通过 Spring Cloud Netflix Ribbon 大大简化了我们在微服务架构中使用客户端负载均衡的成本，服务消费者直接通过调用被 @LoadBalanced 注解修饰过的 RestTemplate 就可以在面向服务的接口调用中自动集成负载均衡功能。

RestTemplate 是 Spring 自带的一个支持 RESTful 风格的 API 调用工具类，@LoadBalanced 注解就是用来给 RestTemplate 添加标记，以便使用负载均衡客户端 LoadBalancerClient 来配置它。LoadBalancerClient 的定义比较简单，只有三个方法。

- ServiceInstance choose(String serviceId)

选择服务实例，根据传入的服务名 serviceId，从负载均衡器中挑选一个对应服务的实例。



- T execute(String serviceId, LoadBalancerRequest request)

执行服务调用，使用从负载均衡器中挑选出的服务实例来执行请求内容。

- URI reconstructURI(ServiceInstance instance, URI original)

构建服务 URI，为系统构建一个合适的 host:port 形式的 URI。使用负载均衡选择的 Service Instance 信息重新构造访问 URI，也就是用服务实例的 host 和 port 来加上服务的路径来构造一个可供访问的真正服务。

当发起一个调用时，负载均衡针对输入的 serviceId 选择一个服务实例，最终调用的也就是 Netflix Ribbon 中 ILoadBalancer 接口所定义的 chooseServer 方法。

### 6.3.4 Spring Cloud Netflix Hystrix 与服务容错

Spring Cloud Netflix Hystrix 同样基于 Netflix Hystrix 实现服务容错，本节也将主要围绕 Netflix Hystrix 进行介绍。

Hystrix (<https://github.com/Netflix/Hystrix/>) 是 Netflix 开源的一款针对分布式系统的延迟和容错库，可用来隔离分布式服务故障，确保系统可用性。

在 Hystrix 中，HystrixCommand 是重中之重，在 Hystrix 的整个机制中，涉及依赖边界的地方，都是通过这个 Command 模式进行调用。我们在使用 HystrixCommand 时需要提供它的子类，子类要实现的方法主要有两个，一个是 run 方法，用于实现依赖的业务逻辑，或者说是实现微服务之间的调用；另一个就是 getFallBack 方法，实现服务降级处理逻辑。

Spring Cloud 内部集成了 Netflix Hystrix 框架，提供@EnableCircuitBreaker 注解构建 Hystrix 客户端。实现过程中，使用 Hystrix 命令模式封装依赖逻辑并提供 Fallback 方法实现降级策略，当 Hystrix 中的业务逻辑发生异常时，将自动切换到 fallback 中的降级逻辑。

#### 1. Hystrix 服务隔离

针对服务隔离，Hystrix 组件提供了两种解决方案，即线程池（Thread Pool）隔离和信号量（Semaphore）隔离，两种隔离方式都限制对共享资源的并发访问量。

##### （1）线程池隔离

Hystrix 线程池运行时效果如图 6-12 所示，可以看到它为不同的服务单独提供了独立的线程池，这与 5.3.3 节中所阐述的线程隔离机制保持一致。



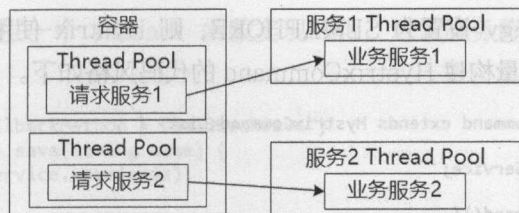


图 6-12 Hystrix 线程池示意图

在使用方式上，我们可以引入 Hystrix 依赖，并使用命令模式封装依赖逻辑。例如，使用线程池构建 HystrixCommand 的代码风格如下所示。

```
public class GetOrderCommand extends HystrixCommand<List> {  
    OrderService orderService;  
  
    public GetOrderCommand(String name){  
        super(Setter.withGroupKey(  
            HystrixCommandGroupKey.Factory.asKey("ThreadPoolTestGroup"))  
            .andCommandKey(HystrixCommandKey.Factory.asKey("testCommandKey"))  
            .andThreadPoolKey(HystrixThreadPoolKey.Factory.asKey(name))  
            .andCommandPropertiesDefaults(  
                HystrixCommandProperties.Setter()  
                    .withExecutionTimeoutInMilliseconds(5000)  
            ).andThreadPoolPropertiesDefaults(  
                HystrixThreadPoolProperties.Setter()  
                    .withMaxQueueSize(10)  
                    .withCoreSize(2)  
            )  
        );  
    }  
  
    @Override  
    protected List run() throws Exception {  
        return orderService.getOrderList();  
    }  
}
```

Hystrix 中可以通过 HystrixThreadPoolProperties 配置线程池参数。例如，coreSize 配置核心线程池大小的最大值，keepAliveTimeMinutes 配置线程池中空闲线程生存时间，maxQueueSize 配置线程池队列大小的最大值等。

线程池的主要缺点就是它增加了计算的开销，每个业务请求（被包装成命令）在执行的时候，会涉及请求排队，调度和上下文切换。对于不依赖网络访问的服务（如只依赖内存缓存），就不适合用线程池隔离技术，而是推荐采用信号量隔离。

## （2）信号量隔离

线程池隔离和信号量隔离的主要区别在于线程池方式下业务请求线程和执行依赖服务的线程不是同一个线程，而信号量方式下业务请求线程和执行依赖服务的线程是同一个线程。

Hystrix 同样提供了 HystrixCommandProperties 用于配置命令的一些参数，将属性



execution.isolation.strategy 设置为 SEMAPHORE，则 Hystrix 使用信号量而不是默认的线程池来做隔离。使用信号量构建 HystrixCommand 的代码风格如下。

```
public class GetOrderCommand extends HystrixCommand<List> {  
    OrderService orderService;  
  
    public GetOrderCommand(){  
        super(Setter.withGroupKey(  
            HystrixCommandGroupKey.Factory.asKey("ThreadPoolTestGroup"))  
            .andCommandPropertiesDefaults(HystrixCommandProperties.Setter()  
                .withExecutionIsolationStrategy(ExecutionIsolationStrategy.SEMAPHORE)));  
    }  
  
    @Override  
    protected List run() throws Exception {  
        return orderService.getOrderList();  
    }  
}
```

### （3）服务分组

结合 5.3.4 节中介绍的服务分组概念，Hystrix 可以粗粒度实现隔离，也可以细粒度实现隔离，包括服务分组+线程池、服务分组+服务+线程池和混合实现等三种模式。

- 服务分组+线程池模式

服务分组+线程池模式是一种粗粒度实现，一个服务分组或系统配置一个隔离线程池即可。可以不配置线程池名称，也可以将相同分组的线程池名称配置为一样。

- 服务分组+服务+线程池模式

服务分组+服务+线程池模式则是一种细粒度实现，一个服务分组中的每一个服务配置一个隔离线程池，为不同的命令实现配置不同的线程池名称即可。

- 混合模式

在混合实现模式下，一个服务分组配置一个隔离线程池，然后对重要服务单独设置隔离线程池。

在 Hystrix 中实现以上服务分组机制的方式是在初始化 HystrixCommand 时设置相应的键值。Hystrix 提供了用于配置全局唯一标识服务分组名称的 HystrixCommandGroupKey，相同分组的服务会聚合在一起；HystrixCommandKey 则用于配置全局唯一标识服务的名称，可以为这个服务起一个名字来唯一识别该服务；HystrixThreadPoolKey 用于配置全局唯一标识线程池的名称，相同线程池名称代表使用的是同一个线程池。

## 2. Hystrix 服务降级和熔断

### （1）服务降级

Hystrix 在服务调用失败（异常、拒绝、超时、短路）时都会执行 fallback 逻辑。在开发



过程上，我们只需要提供一个 fallback 函数实现即可，对于服务降级而言，示例代码如下所示。

```
@HystrixCommand(fallbackMethod = "fallbackSave")
public List<Person> save(String name) {
    return personService.save(name);
}

public List<Person> fallbackSave(String name){
    List<Person> list = new ArrayList<>();
    Person p = new Person(name+"failed to save.");
    list.add(p);
    return list;
}
```

服务降级的整体执行流程如图 6-13 所示，Command 会调用 run()方法，如果 run()方法超时或者抛出异常并且启用了降级处理机制，则调用 getFallback()方法进行降级。

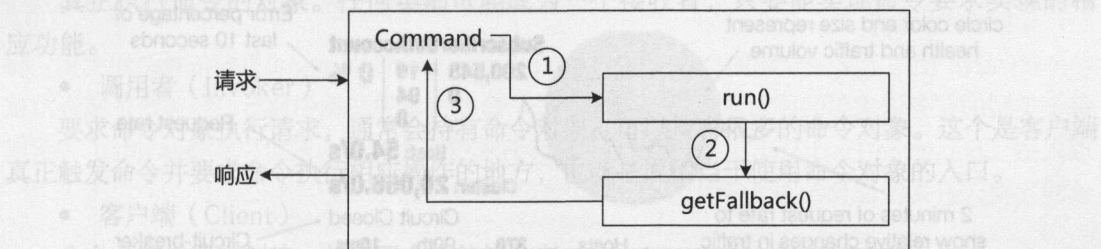


图 6-13 服务降级流程

## （2）服务熔断

Hystrix 提供了熔断实现，熔断后会自动降级处理，如图 6-14 所示。

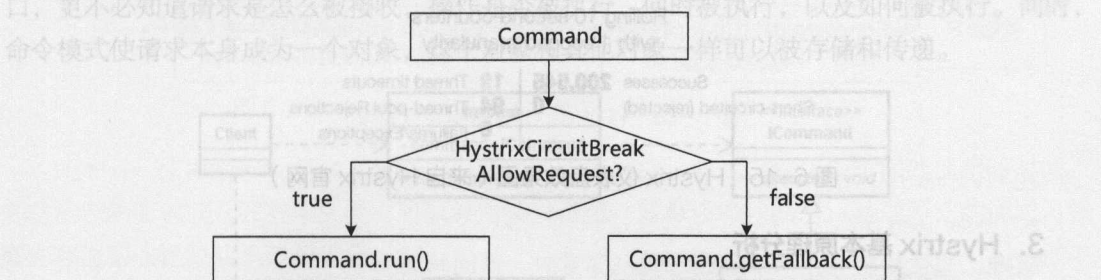


图 6-14 服务熔断流程

Command 首先调用 HystrixCircuitBreaker 的 allowRequest 判断是否熔断了，如果没有熔断，则执行 run()方法正常处理，如果熔断了，则直接调用降级方法 getFallback()方法降级处理。



在 5.3.6 节中我们知道熔断器有三个状态，其中打开和半打开会导致触发熔断机制，表现在流程上就是看图 6-14 中 HystrixCircuitBreaker 的 allowRequest 判断是否成立，而该判断取决于服务访问的失败率。在 Hystrix 中异常、超时、线程池拒绝、信号量拒绝数量总和就是失败总数，通过失败总数与访问总数之间的对比决定失败率，当该失败率超过失败率阈值时就会触发熔断器的打开状态。当熔断处于打开状态后，不能一直熔断下去，需要在一个时间窗口后进行重试，这种状态就是半打开。Hystrix 允许在一定的窗口时间内进行一次重试，重试成功则闭合熔断开关，否则熔断开关还是处于打开状态。

### （3）仪表盘

Hystrix 还提供了一个仪表盘工具用于监控各个 HystrixCommand 的各种运行时状态，这些运行时状态包括请求量、错误率、熔断器状态等各项指标。Hystrix 仪表盘的效果图参考图 6-15。

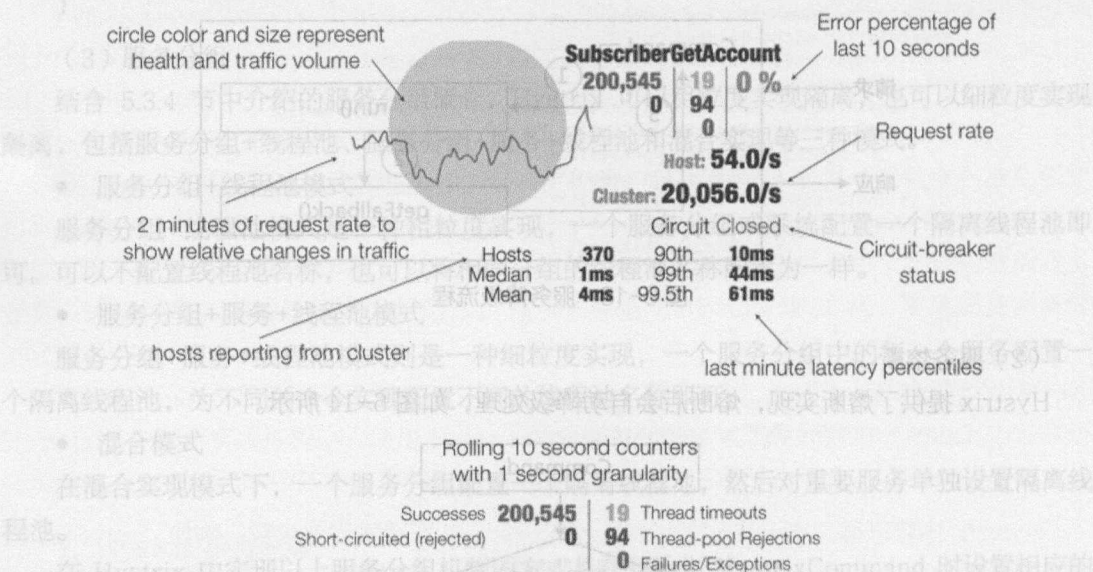


图 6-15 Hystrix 仪表盘效果图（来自 Hystrix 官网）

## 3. Hystrix 基本原理分析

介绍完 Hystrix 提供的核心功能之后，本小节将从设计和实现角度出发分析 Hystrix 的基本原理。

### （1）命令模式与 HystrixCommand

Hystrix 通过命令（Command）模式将每个类型的业务请求封装成对应的命令请求。在具体分析 Hystrix 原理之前，我们先来回顾一下命令模式。



在软件系统中,行为请求者与行为实现者通常是一种紧耦合的关系,但某些场合,如需要对行为进行记录、撤销或重做、事务等处理时,这种无法抵御变化的紧耦合的设计就不太合适。命令模式中,将一个请求封装为一个命令对象,从而可以使用不同的请求对客户进行参数化。同时,对请求排队或记录请求日志,以及支持可撤销的操作。

命令模式的主要组件如下。

- 抽象命令 (Command)

定义命令的接口,声明执行的方法。

- 具体命令 (ConcreteCommand)

具体命令,实现要执行的方法。它通常是“虚”的实现,一般会有命令接收者,并调用接收者的功能来完成命令要执行的操作。

- 接收者 (Receiver)

真正执行命令的对象。任何类都可能成为一个接收者,只要能实现命令要求实现的相应功能。

- 调用者 (Invoker)

要求命令对象执行请求,通常会持有命令对象,可以持有很多的命令对象。这个是客户端真正触发命令并要求命令执行相应操作的地方,也就是说相当于使用命令对象的入口。

- 客户端 (Client)

命令由客户端来创建,并设置命令的接收者。

命令模式的结构图参考图 6-16。该模式的本质是对命令进行封装,将发出命令与执行命令的责任分开。每一个命令都是一个操作:请求的一方发出请求,要求执行一个操作;接收的一方收到请求,并执行操作。将请求方和接收方独立开来,使得请求方不必知道接收方的接口,更不必知道请求是怎么被接收,操作是否被执行、何时被执行,以及如何被执行。同时,命令模式使请求本身成为一个对象,这个对象和其他对象一样可以被存储和传递。

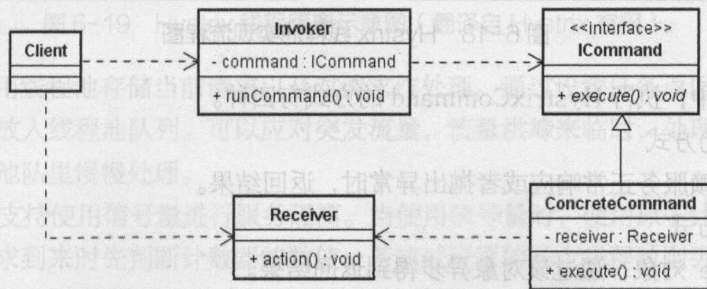


图 6-16 命令模式结构图

命令模式的关键在于引入了抽象命令接口,且发送者针对抽象命令接口编程,只有实现



了抽象命令接口的具体命令才能与接收者相关联。而 Hystrix 提供的基于命令模式进行封装的 HystrixCommand 就体现了这一关键点。HystrixCommand 的结构如图 6-17 所示, 通过实现 HystrixCommand 提供的 run() 方法我们能实现自定义的命令, 并放到 Hystrix 执行环境中供运行时使用, 从而达到命令执行者和接收者之间的分离。

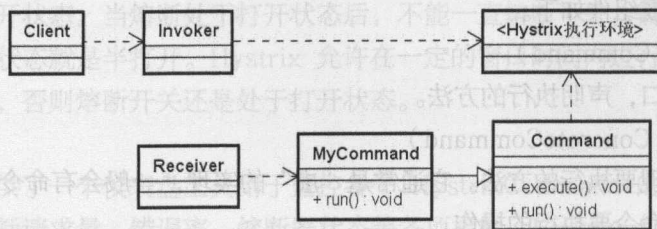


图 6-17 HystrixCommand 结构图

## (2) 线程隔离

Hystrix 使用上文中介绍的命令模式 HystrixCommand(Command) 包装依赖调用逻辑, 将每个类型的业务请求封装成对应的命令请求, 每个命令在单独线程中执行。创建好的线程池被放入到 ConcurrentHashMap 中, 当第二次查询请求过来的时候, 则可以直接从 Map 中获取该线程池。具体流程参考图 6-18。

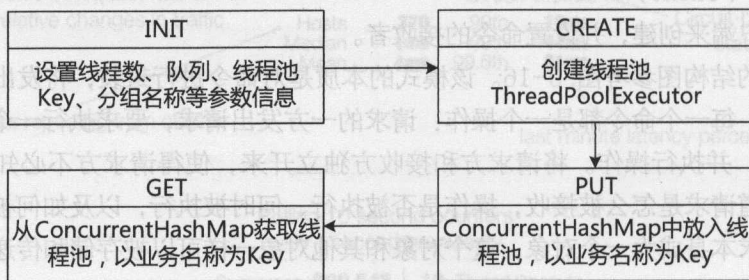


图 6-18 Hystrix 线程池实现流程图

在 Hystrix 中, 执行 HystrixCommand 的方式有四种。

- execute() 方式

阻塞, 当依赖服务正常响应或者抛出异常时, 返回结果。

- queue() 方式

返回 Future 对象, 通过该对象异步得到返回结果。

- observe() 方式

返回 Observable 对象, 立即发出请求, 在依赖服务正常响应或者抛出异常时, 通过注册的 Subscriber 得到返回结果。

非卖品!! 严禁 (销售和上传互联网平台) !! 仅供对书籍质量进行鉴定甄别! 是否为购买正版实体书提供依据!!



• toObservable()方式

返回 Observable 对象,但只有在订阅该对象时,才会发出请求,然后在依赖服务正常响应或者抛出异常时,通过注册的 Subscriber 得到返回结果。

Hystrix 中存在两种基本 Command,即 HystrixCommand 和 HystrixObservableCommand。以上四种执行方式中,execute()和 queue()是在 HystrixCommand 中,而 observe()和 toObservable()则是在 HystrixObservableCommand 中。从底层实现来讲,HystrixCommand 其实也是利用 Observable 实现的,内部大量使用了 RxJava。RxJava 是一个响应式编程框架,读者有兴趣可以参考其官方网站(<http://reactivex.io/>)上的介绍。

HystrixCommand 执行过程中,执行依赖代码的线程与请求线程(如 Tomcat 线程)分离,请求线程可以自由控制离开的时间,这也是我们通常说的异步编程,Hystrix 是结合 RxJava 来实现的异步编程。通过设置线程池大小来控制并发访问量,当线程饱和的时候可以拒绝服务,防止依赖问题扩散(见图 6-19)。



图 6-19 Hystrix 线程隔离示意图 (翻译自 Hystrix 官网)

Hystrix 使用线程池存储当前请求以及对请求作处理。通过设置任务返回处理超时时间,并将堆积的请求放入线程池队列。可以应对突发流量,流量洪峰来临时,处理不完的请求可将数据存储在队列里慢慢处理。

Hystrix 还支持使用信号量进行服务隔离。当使用信号量时,使用原子计数器来记录当前运行线程数,请求到来时先判断计数器的数值,若超过设置的最大线程数则丢弃该类型的新请求,若不超过则执行计数器+1,请求返回时执行计数器-1。信号量隔离无法应对突发流量。关于线程池和信号量两种隔离机制,表 6-4 做了一个全面的总结。



表 6-4

Hystrix 两种隔离模式对比

对比项	线程池隔离	信号量隔离
线程	与调用线程隔离	与调用线程相同
成本	排队、调度和上下文切换等开销	无线程切换，成本较低
异步	支持	不支持
并发	基于最大线程池大小	基于最大信号量上限

### (3) 熔断机制

Hystrix 通过 `HystrixCircuitBreaker` 接口定义了服务熔断器机制。`HystrixCircuitBreaker` 接口只有三个方法，分别介绍如下。

- `allowRequest()`

每个 Hystrix 命令的请求都通过它判断是否可被执行。先根据配置对象 `properties` 中的断路器判断强制打开或关闭属性是否被设置，如果强制打开，则直接返回 `false` 拒绝请求。如果强制关闭则允许所有请求，但是同时会调用下面的 `isOpen()` 方法来执行断路器的计算逻辑，用来模拟断路器打开和关闭的行为。

- `isOpen()`

返回当前断路器是否打开。如果断路器打开标识为 `true`，则直接返回 `true`。如果不是，则从度量指标对象 `metrics` 中获取运行时统计对象做进一步判断。

- `markSuccess()`

关闭断路器。该方法在断路器处于“半开”状态下使用，如果 Hystrix 命令调用成功，通过调用该方法可以将打开的断路器关闭，并重置度量指标对象。

结合命令模式、线程隔离和熔断机制的 Hystrix 整体运行流程见图 6-20，包括以下主要步骤。

(A) 构建 Hystrix 的 `Command` 对象，调用执行方法。

(B) Hystrix 检查当前服务的熔断器开关是否开启，若开启，则执行降级服务 `getFallback` 方法。

(C) 若熔断器开关关闭，则 Hystrix 检查当前服务的线程池是否能接收新的请求，若线程池已满，则执行降级服务 `getFallback` 方法。

(D) 若线程池接受请求，则 Hystrix 开始执行服务调用具体逻辑 `run` 方法。

(E) 若服务执行失败，则执行降级服务 `getFallback` 方法，并将执行结果上报 `Metrics` 更新服务健康状况。

(F) 若服务执行超时，则执行降级服务 `getFallback` 方法，并将执行结果上报 `Metrics` 更新服务健康状况。



- (G) 若服务执行成功，返回正常结果。
- (H) 若服务降级方法 getFallback 执行成功，则返回降级结果。
- (I) 若服务降级方法 getFallback 执行失败，则抛出异常。

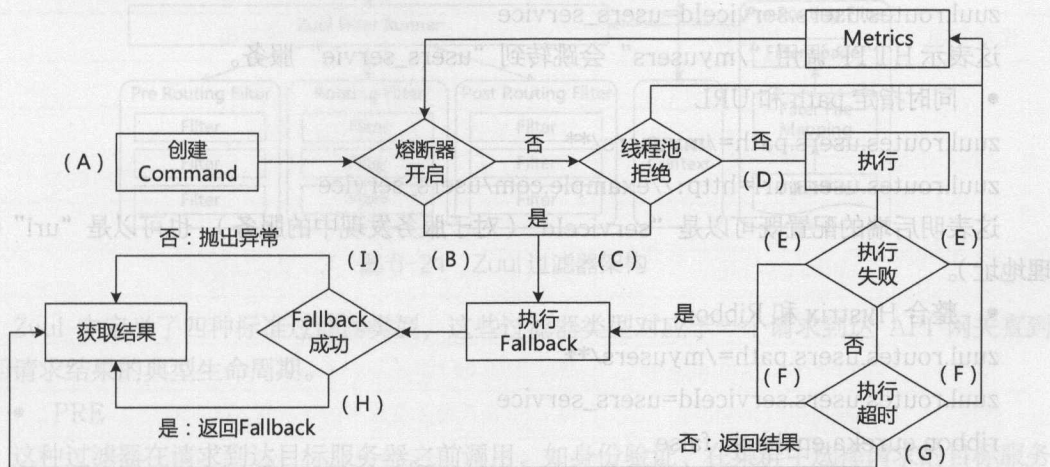


图 6-20 Hystrix 整体运行流程图

### 6.3.5 Spring Cloud Netflix Zuul 与 API 网关

在 4.5 节中我们介绍了 API 网关在微服务架构中的作用和应该具备的诸如业务路由、安全管理、日志记录、访问控制等核心功能。作为 API 网关的一种实现方式，Spring Cloud Netflix Zuul 为我们提供了类似功能。Spring Cloud Netflix Zuul 同样基于 Netflix 旗下的 Zuul 组件，本节中我们也从 Netflix Zuul 开始讲起。

#### 1. Netflix Zuul 路由机制

正如我们所想象的 API 网关，Zuul 为我们提供动态路由、监控、安全等边缘服务，并提供了 @EnableZuulProxy 注解支持代码中嵌入 Zuul 反向代理服务器。对于 API 网关而言，最重要的功能就是服务路由。在这里，我们引用 Zuul 官方文件中的几个例子来说 Zuul 的路由机制。

- 自定义指定微服务的访问路径

```
zuul.routes.users=/myusers/**
```

这表示 HTTP 调用 “/myusers” 会转到 “users” 服务。

- 忽略所有微服务，只路由指定微服务

```
zuul.ignoredServices=“*”
```

```
zuul.routes.users=/myusers/**
```



在这个例子中，所有的服务都会被忽略，除了“users”。

- 同时指定微服务的 serviceId 和对应的路径

```
zuul.routes.users.path=/myusers/**
```

```
zuul.routes.users.serviceId=users_service
```

这表示 HTTP 调用“/myusers”会跳转到“users\_service”服务。

- 同时指定 path 和 URL

```
zuul.routes.users.path=/myusers/**
```

```
zuul.routes.users.url=http://example.com/users_service
```

这表明后端的配置既可以是“serviceId”（对于服务发现中的服务），也可以是“url”（物理地址）。

- 整合 Hystrix 和 Ribbon

```
zuul.routes.users.path=/myusers/**
```

```
zuul.routes.users.serviceId=users_service
```

```
ribbon.eureka.enabled: false
```

```
users.ribbon.listOfServers: example.com,google.com
```

要实现 Hystrix 和 Ribbon 整合，需要给这个 serviceId 指定一个 service-route 并配置一个 Ribbon Client。

## 2. Netflix Zuul 过滤器机制

ZuulFilter 是 Zuul 中的核心组件，通过继承该抽象类，覆写几个关键方法就能达到自定义调度请求的效果。ZuulFilter 定义了以下几个主要方法。

- filterOrder

filter 执行顺序，通过数字指定。

- shouldFilter

filter 是否需要执行。

- run

filter 具体实现逻辑。

- filterType

filter 类型，内置主要分为四种，后面会具体展开介绍。

Zuul 提供了一个框架，可以对过滤器进行动态的加载、编译和运行。过滤器之间没有直接的相互通信，他们是通过一个 RequestContext 的静态类来进行数据传递的。RequestContext 类中有 ThreadLocal 变量来记录每个 Request 所需要传递的数据。Zuul 过滤器整体架构参考图 6-21。



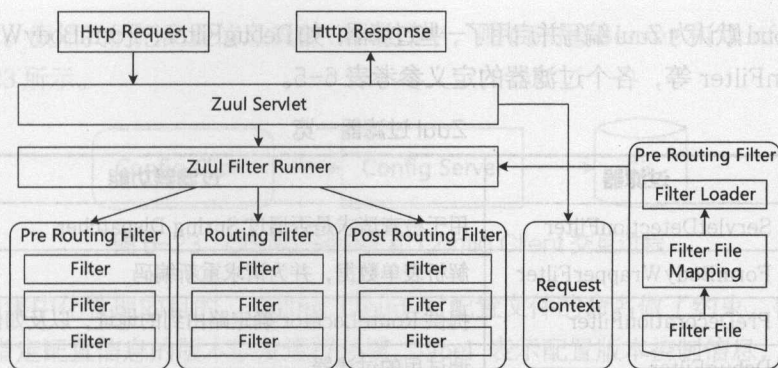


图 6-21 Zuul 过滤器架构

Zuul 中定义了四种标准过滤器类型，这些过滤器类型对应于一个请求到达 API 网关直到返回请求结果的典型生命周期。

- PRE

这种过滤器在请求到达目标服务器之前调用，如身份验证、在集群中选择请求的目标服务器、记录请求日志等。

- ROUTING

在这种过滤器中把用户请求发送给目标服务器。发送给目标服务器的用户请求在这类过滤器中被重新构建，并使用 Apache HttpClient 或者 Netfilx Ribbon 发送给目标服务器。

- POST

这种过滤器在用户请求从目标服务器返回以后执行，比如在返回的响应上添加响应头信息以及做各种统计等。

- ERROR

在其他阶段发生错误时执行该过滤器。

Zuul 请求的生命周期如图 6-22 所示，该图详细描述了各种类型过滤器的执行顺序。

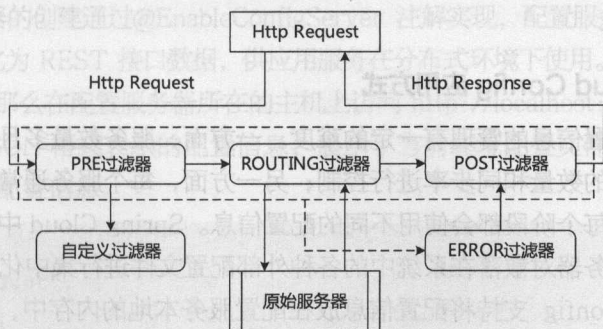


图 6-22 Zuul 过滤器生命周期（翻译自 Zuul 官网）



Spring Cloud 默认为 Zuul 编写并启用了一些过滤器，如 `DebugFilter`、`FormBodyWrapperFilter`、`PreDecorationFilter` 等，各个过滤器的定义参考表 6-5。

表 6-5

Zuul 过滤器一览

生命周期	过滤器	过滤器功能
PRE	<code>ServletDetectionFilter</code>	用于检查请求是否通过 Spring Dispatcher
	<code>FormBodyWrapperFilter</code>	解析表单数据，并为请求重新编码
	<code>PreDecorationFilter</code>	提供 <code>RouteLocator</code> 确定路由到的地址，以及如何路由
	<code>DebugFilter</code>	调试用的过滤器
ROUTING	<code>SendForwardFilter</code>	使用 <code>Servlet RequestDispatcher</code> 转发请求
	<code>RibbonRoutingFilter</code>	使用 <code>Ribbon</code> 、 <code>Hystrix</code> 和可插拔的 HTTP 客户端发送请求
	<code>SimpleHostRoutingFilter</code>	通过 <code>Apache HttpClient</code> 向指定的 URL 发送请求
POST	<code>SendResponseFilter</code>	将 Zuul 所代理的微服务的响应写入当前响应
ERROR	<code>SendErrorFilter</code>	处理有错误的请求响应，默认转发到 <code>/error</code>

我们介绍 Zuul 的方式是先讲路由器，再讲过滤机制。从表面上讲，路由功能负责将请求转发到具体的微服务实例上，而过滤器功能则是对请求的处理过程进行干预。但事实上，路由功能在运行过程中，Zuul 的路由映射和请求转发都是由几个内置的过滤器实现的，其中路由映射主要通过 PRE 类型过滤器完成，它将请求路由与配置的路由规则进行匹配以找到需要转发的目标地址；而请求转发部分则由 ROUTING 类型的过滤器完成，对 PRE 类型过滤器获取的路由地址进行转发，所以过滤器可以说是 Zuul 实现 API 网关功能最为核心的组件。

### 6.3.6 Spring Cloud Config 与配置中心

与前面介绍的各种组件不同，Spring Cloud Config 是 Spring Cloud 家族自己研发的高可用、分布式配置中心。

#### 1. Spring Cloud Config 应用方式

分布式环境下配置信息的管理有一定的难度。一方面，服务数量多且运行在集群环境，意味着需要对配置信息的数量和同步率进行控制；另一方面，每个服务通常分为开发、测试、生产等生命周期阶段，每个阶段都会使用不同的配置信息。Spring Cloud 中配置中心的设计理念就是通过一个配置服务器对散落在系统中的各种外部配置文件进行集中化管理。

Spring Cloud Config 支持将配置信息放在配置服务本地的内存中，也支持放在远程 Git 仓库中。在 Spring Cloud Config 组件中，分两个角色，一个是 `Config Server`，另一个是



Config Client。假如我们把配置信息放在 Git 仓库中，则 Config Server 和 Config Client 交互过程如图 6-23 所示。

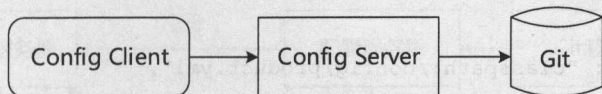


图 6-23 Config Server 和 Config Client 交互过程

为了达到集中化管理的目的，Spring Cloud 对配置文件的命名做了约束，使用 label 和 profile 概念指定配置信息的版本以及运行环境。label 表示配置版本控制信息，默认使用 git master 对应的主线版本；而 profile 中的 dev、prod、test 分别对应着开发、生产和测试环境。配置文件可以使用类如 `/application/profile/label`、`application-profile.yml`、`label/application-profile.yml`、`application-profile.properties`、`label/application-profile.properties` 等命名方式，同时支持 properties 和 yml 两种文件格式。

.yml 文件格式是 YAML (Yet Another Markup Language) 编写的文件格式，YAML 是一种直观的、能够被电脑识别的数据序列化格式，并且容易被人类阅读，容易和脚本语言进行交互，表现形式如下。

```
spring:
  application:
    name: config
  profiles:
    active: native
eureka:
  instance:
    non-secure-port: ${server.port:8888}
    metadata-map:
      instanceId: ${spring.application.name}:${random.value}
  client:
    service-url:
      defaultZone: http://${eureka.host:localhost}:${eureka.port:8761}/eureka/
```

集中式配置服务器的创建通过 `@EnableConfigServer` 注解实现，配置服务器可以将存放在仓库中的配置文件信息转化为 REST 接口数据，供应用服务在分布式环境下使用。假设在 product 系统存在一个 env 环境，那么在配置服务器所在的主机上访问 `http://localhost:8888/product/env` 即可得到类似如下以 JSON 格式表示的配置信息，展示了当前环境配置文件的元数据以及包含在配置文件中的数据库配置信息。

```
{
  "name": "product",
  "profiles": [
    "env"
  ]
}
```



```

    ],
    "label": "master",
    "propertySources": [
      {
        "name": "classpath:/config/product.yml",
        "source": {
          "spring.jpa.database": "MYSQL",
          "spring.datasource.platform": "mysql",
          "spring.datasource.url":
            "jdbc:mysql://127.0.0.1:3306/microservice_product",
          "spring.datasource.username": "root",
          "spring.datasource.password": 120822,
          "spring.datasource.driver-class-name":
            "com.mysql.jdbc.Driver"
        }
      }
    ]
  }
}

```

Config Client 从配置服务器中获取配置信息基于如下的工作流程。

(A) 各个客户端应用启动时，根据配置的应用名{application}、环境名{profile}和分支名{label}，向配置服务器请求配置信息。

(B) 配置服务器根据自己维护的 Git 仓库信息和客户端传递过来的配置定位参数去查找具体的配置信息。

(C) 通过 git clone 命令将找到的配置文件下载到配置服务器本地的文件系统中。

(D) 配置服务器从本地加载配置文件并将这些配置信息读取出来之后以 JSON 格式返回给客户端应用。

(E) 客户端应用在获取配置服务器传递过来的配置信息后加载到本地服务并根据这些配置信息完成相应操作。

配置服务器在执行以上步骤的过程中，通过 git clone 命令将配置信息存放在本地，从而起到缓存作用，即使当 Git 服务器无法访问的时候，客户端应用仍然可以获取并使用配置服务器中的缓存内容。

## 2. Spring Cloud Config 基本架构

(1) Spring Cloud Config 对比 Zookeeper

在 4.6 节中，我们介绍了配置中心的基本模型，同时也给出了基于分布式协调框架



Zookeeper 的实现方案。Spring Cloud Config 所采用的设计思路与 Zookeeper 完全不同，其基本架构如图 6-24 所示。

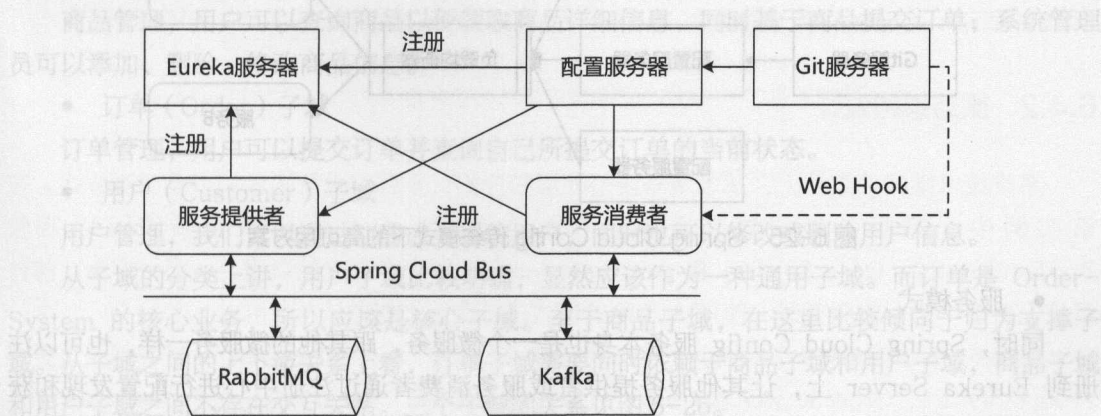


图 6-24 Spring Cloud Config 基本架构

图 6-24 中，从架构上就可以看出 Spring Cloud Config 与 Zookeeper 的主要区别在于以下几个方面。

- 配置的存储方式不同

Zookeeper 是把配置信息保存在 Zookeeper 自身的文件系统中，而 Spring Cloud Config 则是将配置信息保存在 Git/SVN 上。

- 配置的管理方式不同

Spring Cloud Config 就是把配置都当成源码来看待并通过 Git 进行管理。Git 的管理界面，就是配置的管理界面。而基于 Zookeeper 的实现方案一般需要自己开发管理界面。

- 配置变化的通知机制不同

Zookeeper 中的配置变化依赖其自身的事件 Watcher 机制来通知应用；而 Spring Cloud Config 则是依赖 Git 每次 push 后触发的 webhook 回调并发送事件到 Spring Cloud Bus 消息总线，然后由消息总线通知相关的应用。

## (2) Spring Cloud Config 高可用

Spring Cloud Config 实现高可用的方式主要有如下两种。

- 传统模式

在传统模式下，所有的配置服务器统一指向同一个 Git 仓库，这样所有的配置内容就通过基于 Git 的共享文件系统进行维护。而客户端在指定配置服务器地址时，只需要设置配置服务器上层的负载均衡器即可。基于传统模式的高可用方案参考图 6-25。



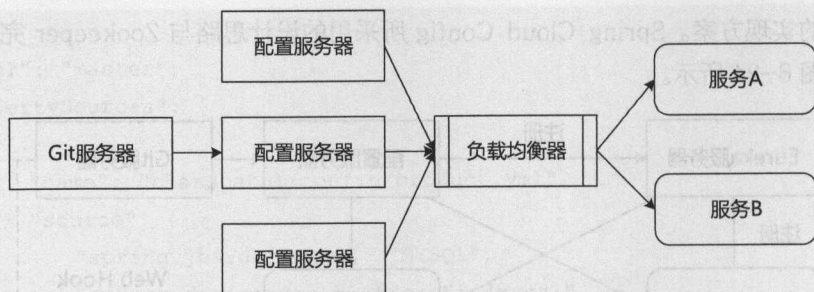


图 6-25 Spring Cloud Config 传统模式下的高可用方案

### • 服务模式

同时，Spring Cloud Config 服务本身也是一个微服务，跟其他的微服务一样，也可以注册到 Eureka Server 上，让其他服务提供者或服务消费者通过注册中心进行配置发现和获取。显然，这种方式比传统模式更加容易维护，因为服务器端的负载均衡和客户端的配置中心指定功能都通过基于 Eureka 的服务治理机制同时得到解决，从而也就间接实现了高可用性。

## 6.4 案例分析

在本章中，我们通过构建一个简单但又完整的系统来展示微服务架构相关的设计理念和常见实现技术。

该系统称为订单系统（Order-System），试图对互联网应用中最常见的订单业务做抽象。现实环境中订单业务可以非常复杂，该案例的目的在于演示从业务领域分析到系统架构设计再到系统实现的整个过程，不在于介绍具体业务逻辑，所以业务领域建模上做了高度抽象，并不代表实际的应用场景。

本章对系统案例的讲解，首先会基于领域驱动思想对案例业务进行领域建模，然后开展微服务架构设计，最后采用本书中介绍的具体工具和框架完成案例的实现。

### 6.4.1 服务建模

按照本书对微服务架构的讨论思路，服务建模是案例分析的第一步。服务建模包括子域与界限上下文的划分以及服务拆分和集成策略的确定。

#### 1. 子域与界限上下文

Order-System 包含的业务场景比较简单，用户首先需要登录，登录成功之后会浏览商品，然后在商品列表中选择所想购买的商品并提交订单。从领域的角度进行分析，我们可以把



该系统分成三个子域。

- 商品 (Product) 子域

商品管理, 用户可以查询商品以便获取商品详细信息, 同时基于商品提交订单; 系统管理员可以添加、删除、修改商品信息。

- 订单 (Order) 子域

订单管理, 用户可以提交订单并查询自己所提交订单的当前状态。

- 用户 (Customer) 子域

用户管理, 我们可以通过注册成为系统用户, 同时也可以修改或删除用户信息。

从子域的分类上讲, 用户子域比较明确, 显然应该作为一种通用子域。而订单是 Order-System 的核心业务, 所以应该是核心子域。至于商品子域, 在这里比较倾向于归为支撑子域。从子域之间的上下游关系上看, 订单子域需要同时依赖于商品子域和用户子域, 商品子域和用户子域之间不存在交互关系。三个子域的关系见图 6-26。

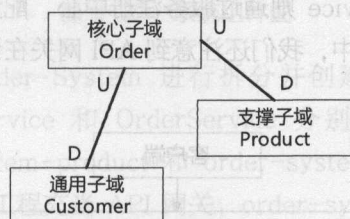


图 6-26 Order-System 子域与界限上下文示意图

## 2. 服务拆分与集成

为了简单起见, 我们对每一个子域都提取一个微服务作为示例。基于以上分析, 我们可以把 Order-System 简单划分成三个微服务, 即 ProductService、OrderService 和 CustomerService, 图 6-27 展示了 Order-System 的基本架构。在图 6-27 中, 三个微服务之间需要基于 REST 进行跨服务之间的交互。

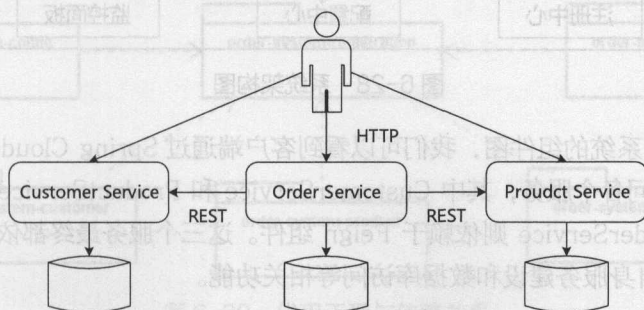


图 6-27 Order-System 服务模型



除了业务建模，图 6-27 还展示了非常重要的一条建模原则，即数据分离原则，每个服务的业务数据只维护在这个服务内部，服务与服务之间不通过数据进行集成。对于 Order-System 而言，Order 对象中会保存 Product 和 Customer 对象的主键信息，但仅此而已。

## 6.4.2 服务架构设计

系统技术架构的细化上，基本策略还是分层和分割，并合理设置子系统以及交互方式。在该案例中，微服务架构的应用体现在两个方面。一方面，后台系统可以抽象 ProductService、OrderService 和 CustomerService 等三个独立的微服务；另一方面，前台系统面对多个后台微服务时，使用 API 网关模式进行服务的有效访问。

案例系统架构如图 6-28 所示，客户端通过 API 网关连接到后台系统，API 网关负责将 ProductService、OrderService 和 CustomerService 整合到一个业务链路，而 ProductService、OrderService 和 CustomerService 则通过服务注册中心、配置管理中心和监控面板进行分布式环境的系统整合。在图 6-28 中，我们还注意到 API 网关在访问各个服务时都将使用熔断器机制确保服务熔断和降级。

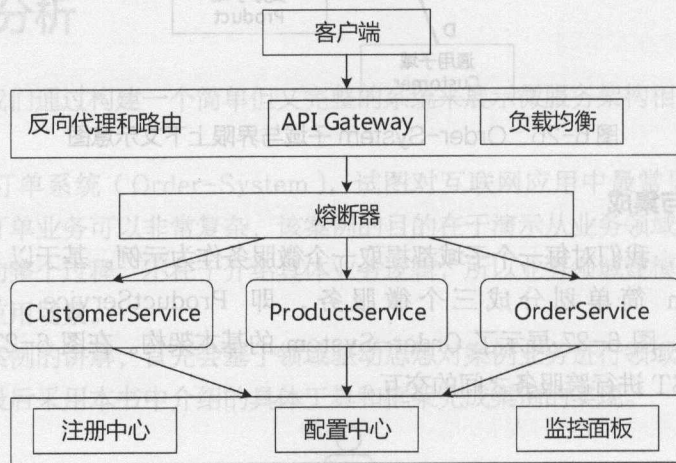


图 6-28 系统架构图

图 6-29 展示了系统的组件图，我们可以看到客户端通过 Spring Cloud Netflix Zuul 提供的 API 网关功能访问各个服务，其中 CustomerService 和 ProductService 使用 Ribbon 的负载均衡机制，而 OrderService 则依赖于 Feign 组件。这三个服务最终都依赖配置服务器提供的配置信息，完成自身服务建设和数据库访问等相关功能。



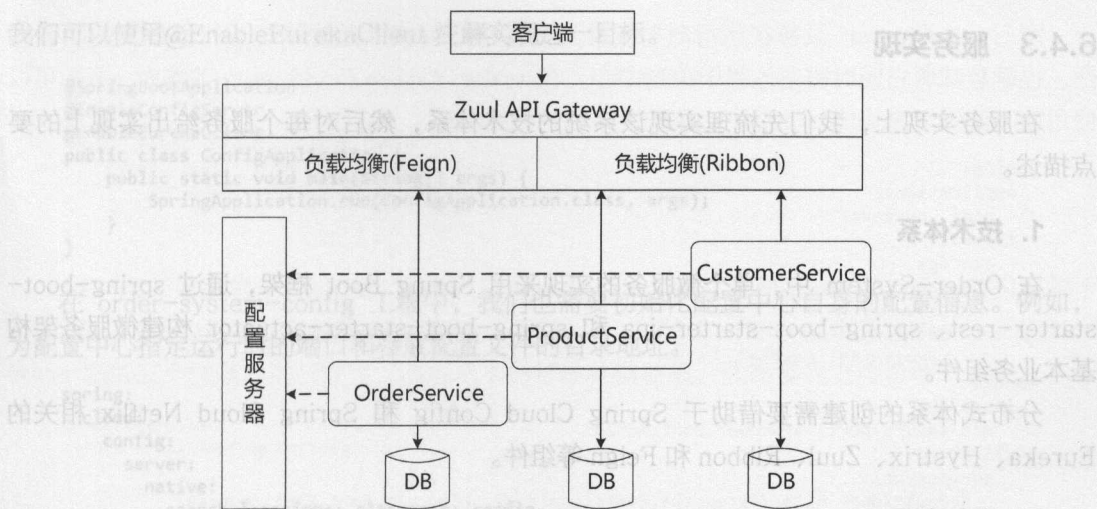


图 6-29 系统组件图

我们结合系统架构对 Order-System 进行拆分并创建代码工程，首先三个微服务 CustomerService、ProductService 和 OrderService 分别独立成三个代码工程 order-system-customer、order-system-product 和 order-system-order。系统存在一个面向前端的 order-system-gateway 工程充当 API 网关，order-system-config 作为集中式的配置服务器通常也单独作为一个工程进行管理，我们还添加了 order-system-monitor 作为系统监控服务。而在基于 Spring Cloud 的开发模式中，上述这些工程均依赖于注册中心 order-system-registration。系统代码工程依赖关系如图 6-30 所示。

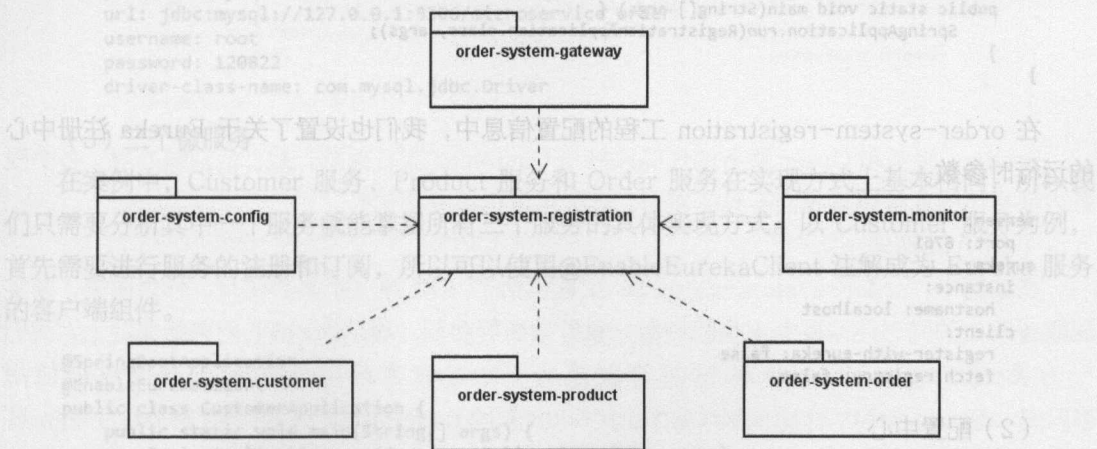


图 6-30 代码工程与依赖关系



### 6.4.3 服务实现

在服务实现上,我们先梳理实现该系统的技术体系,然后对每个服务给出实现上的要点描述。

#### 1. 技术体系

在 Order-System 中,单个微服务的实现采用 Spring Boot 框架,通过 spring-boot-starter-rest、spring-boot-starter-jpa 和 spring-boot-starter-actuator 构建微服务架构基本业务组件。

分布式体系的创建需要借助于 Spring Cloud Config 和 Spring Cloud Netflix 相关的 Eureka、Hystrix、Zuul、Ribbon 和 Feign 等组件。

#### 2. 服务实现

在本节中,我们将对几个主要的组件各自实现方式做简要说明。Order-System 中各个工程的代码实现以及工程之间的依赖关系请参考:<https://github.com/tianminzheng/order-system-microservice>。

##### (1) 注册中心

注册中心的实现非常简单,通过使用@EnableEurekaServer 注解,order-system-registration 工程就能启动 Eureka 服务器。

```
@SpringBootApplication
@EnableEurekaServer
public class RegistrationApplication {
    public static void main(String[] args) {
        SpringApplication.run(RegistrationApplication.class, args);
    }
}
```

在 order-system-registration 工程的配置信息中,我们也设置了关于 Eureka 注册中心的运行时参数。

```
server:
  port: 8761
eureka:
  instance:
    hostname: localhost
  client:
    register-with-eureka: false
    fetch-registry: false
```

##### (2) 配置中心

配置中心的实现依赖于 Spring Cloud Config 组件,通过@EnableConfigServer 注解即可启动配置服务器。同时,配置服务器作为服务的提供者,同样需要注册到 Eureka 服务器中,



我们可以使用@EnableEurekaClient 注解实现这一目标。

```
@SpringBootApplication
@EnableConfigServer
@EnableEurekaClient
public class ConfigApplication {
    public static void main(String[] args) {
        SpringApplication.run(ConfigApplication.class, args);
    }
}
```

在 order-system-config 工程中,我们也需要初始化配置中心自身的配置信息。例如,为配置中心指定运行时的端口和存放配置文件的目录地址。

```
spring:
  cloud:
    config:
      server:
        native:
          search-locations: classpath:/config
server:
  port: 8888
```

基于集中化配置管理策略,我们在配置中心存放着各个服务可能需要的配置信息。例如,案例中的 order-system-order 工程需要访问数据库以便保存用户订单信息,那就可以把数据库访问的相关元数据通过配置中心进行统一管理。其他的各个微服务也可以采用同样的配置信息管理策略。

```
spring:
  jpa:
    database: MYSQL
datasource:
  platform: mysql
  url: jdbc:mysql://127.0.0.1:3306/microservice_order
  username: root
  password: 120822
  driver-class-name: com.mysql.jdbc.Driver
```

### (3) 三个微服务

在案例中, Customer 服务、Product 服务和 Order 服务在实现方式上基本相同,所以我们只需要分析其中一个服务就能掌握所有三个服务的具体实现方式。以 Customer 服务为例,首先需要进行服务的注册和订阅,所以可以使用@EnableEurekaClient 注解成为 Eureka 服务的客户端组件。

```
@SpringBootApplication
@EnableEurekaClient
public class CustomerApplication {
    public static void main(String[] args) {
        SpringApplication.run(CustomerApplication.class, args);
    }
}
```

我们在图 6-29 所示的系统组件图中看到, Customer 服务和 Product 服务使用 Ribbon 组



同时，Customer 服务在运行时需要使用配置中心提供的配置信息进行数据库访问等操作，也就意味着它同时需要依赖于配置中心。在 order-system-customer 工程中，我们可以使用如下配置项来完成对注册中心和配置中心的无缝集成。

```
spring:
  application:
    name: customer
  cloud:
    config:
      enabled: true
      discovery:
        enabled: true
        service-id: config
    eureka:
      instance:
        non-secure-port: ${server.port:8082}
      client:
        service-url:
          defaultZone: http://${eureka.host:localhost}:${eureka.port:8761}/eureka/
```

Customer 服务的一个功能是通过用户提交订单时所带有的用户信息来判断该用户是否合法，所以在 Customer 服务中我们将暴露一个根据 CustomerId 获取 Customer 详细信息的访问入口，并通过 RESTful 风格将该入口开放给其他服务使用。使用 Spring Boot 构建 RESTful 风格的 API 访问入口参考如下方式。

```
@RestController
public class CustomerController {

    @Autowired
    CustomerService customerService;

    @RequestMapping(value = "/{customerId}", method = RequestMethod.GET)
    public Customer getCustomer(@PathVariable Long customerId) {
        Customer customer = customerService.getCustomerById(customerId);
        return customer;
    }
}
```

Product 服务和 Order 服务的代码风格与 Customer 服务保持一致，因篇幅有限，各个服务内部具体的业务逻辑我们不具体展开，读者可自行参考源代码。

#### （4）API 网关

在案例中，order-system-gateway 工程包括了 API 网关的具体实现。作为网关，在实现服务路由的同时还需要考虑负载均衡、服务熔断等功能。所以在 API 网关的实现上，我们一方面通过 @EnableEurekaClient 注解成为 Eureka 客户端，另一方面也可以使用 @EnableFeignClients、@EnableCircuitBreaker 和 @EnableZuulProxy 等注解启动 Feign 客户端、熔断器和 Zuul 负载均衡机制。



```
@SpringBootApplication
@EnableEurekaClient
@EnableFeignClients
@EnableCircuitBreaker
@EnableZuulProxy
public class GatewayApplication {
    public static void main(String[] args) {
        SpringApplication.run(GatewayApplication.class, args);
    }
}
```

API 网关所提供的访问入口直接面向客户端，我们通过使用 Spring Boot 来构建基于 RESTful 风格的服务接口机制。通过 API 网关，我们把 Customer 服务、Product 服务和 Order 服务三个微服务进行整合，围绕基本的订单处理流程，代码实现上可参考如下。

```
@RestController
public class GatewayController {

    @Autowired
    private CustomerHystrixService customerHystrixService;

    @Autowired
    private OrderHystrixService orderHystrixService;

    @Autowired
    private ProductHystrixService productHystrixService;

    @RequestMapping("/gateway")
    public ResultMessage gateway(Long customerId, String productCode) throws Exception {
        Customer customer = customerHystrixService.getCustomer(customerId);
        if(customer == null || customer.getDescription().contains("fail")) {
            return ResultMessageBuilder.build(false,
                "The customer for " + customerId + " is not existed!");
        }

        Product product = productHystrixService.getProduct(productCode);
        if(product == null || product.getDescription().contains("fail")) {
            return ResultMessageBuilder.build(false,
                "The product for " + productCode + " is not existed!");
        }

        Order order = new Order();
        order.setCustomerId(customerId);
        order.setProductCode(productCode);
        order.setCreateTime(new Date());
        List<Order> orders = orderHystrixService.save(order);

        return ResultMessageBuilder.build(orders);
    }
}
```

在上述代码中，我们发现 API 网关并不是直接依赖 CustomerService、ProductService 和 OrderService 这些原始的微服务定义，而是分别基于 CustomerHystrixService、ProductHystrixService 和 OrderHystrixService 完成了订单业务流程。从命名上看，我们不难想象这三个服务之中都添加了 Hystrix 支持。

我们在图 6-29 所示的系统组件图中看到，Customer 服务和 Product 服务使用 Ribbon 组



件实现负载均衡, 而 Order 服务则依赖于 Feign 组件。其中, 基于 Ribbon 组件实现负载均衡的方法可以采用 RestTemplate。例如, 在 CustomerHystrixService 中, 我们看到系统通过 RestTemplate 对 Customer 进行远程访问, 并提供了 fallback 方法来完成回退操作。同样的代码风格也体现在了 ProductHystrixService 中。

```
@Service
public class CustomerHystrixService {
```

```
    @Autowired
    RestTemplate restTemplate;
```

```
    @HystrixCommand(fallbackMethod = "fallbackCustomer")
```

```
    public Customer getCustomer(Long customerId) {
        return restTemplate.getForObject("http://Customer/" +
            customerId, Customer.class);
    }
```

```
    public Customer fallbackCustomer(Long customerId){
        Customer customer = new Customer();
        customer.setDescription(customerId + ": Customer service failed!");
        return customer;
    }
```

而 OrderHystrixService 则采用 Feign 组件来完成负载均衡。我们设计一个订单访问接口定义, 并通过添加@FeignClient 注解, 我们就能使用该接口定义来完成对 Order 服务的远程访问。

```
@FeignClient("order")
```

```
public interface OrderService {
```

```
    @RequestMapping(method = RequestMethod.POST, value = "/save",
        produces = MediaType.APPLICATION_JSON_VALUE,
        consumes = MediaType.APPLICATION_JSON_VALUE)
```

```
    @ResponseBody
```

```
    List<Order> save(@RequestBody Order order);
```

```
}
```

而在 OrderHystrixService 中, 我们直接注入该接口就可以实现负载均衡。

```
@Service
```

```
public class OrderHystrixService {
```

```
    @Autowired
```

```
    OrderService orderService;
```

```
    @HystrixCommand(fallbackMethod = "fallbackSave")
```

```
    public List<Order> save(Order order) {
        return orderService.save(order);
    }
```

```
    public List<Order> fallbackSave(Order order){
```

```
        List<Order> list = new ArrayList<>();
```

```
        Order o = new Order();
```

```
        o.setProductCode(order.getProductCode()+" : Order service failed!");
```

```
        list.add(o);
```

```
        return list;
```

```
    }
```

```
}
```



### (5) 服务监控

服务监控上,可以使用 Hystrix Dashboard 查看单个服务的运行状态,而要想看系统的 Hystrix Dashboard 数据就需要用到 Hystrix Turbine。Hystrix Turbine 将每个服务的 Hystrix Dashboard 数据进行了整合。Hystrix Dashboard 和 Hystrix Turbine 的使用非常简单,只需要引入相应的依赖和加上注解和配置即可。在 order-system-monitor 工程中我们就通过@EnableHystrixDashboard 和@EnableTurbine 注解启动了服务监控机制。

```
@SpringBootApplication
@EnableEurekaClient
@EnableHystrixDashboard
@EnableTurbine
public class MonitorApplication {

    public static void main(String[] args) {
        SpringApplication.run(MonitorApplication.class, args);
    }
}
```

## 6.5 本章小结

本章基于第4章和第5章中所介绍的微服务架构实现基础组件和关键要素,给出了这些组件和要素的具体实现技术和工具。

本章首先讨论了如何对微服务架构实现技术进行正确选型的问题,我们给出了技术选型的参考标准,并对微服务实现框架进行了对比,最终确定使用 Spring Cloud 作为实现微服务架构的主要开发工具。

Spring Cloud 基于 Spring Boot,在对 Spring Boot 核心原理进行分析之后,我们重点介绍了 Spring Cloud 的部分组件,包括实现服务治理的 Spring Cloud Netflix Eureka 组件、实现负载均衡的 Spring Cloud Netflix Ribbon 组件、实现服务容错的 Spring Cloud Netflix Hystrix 组件、实现 API 网关的 Spring Cloud Netflix Zuul 组件以及实现配置中心的 Spring Cloud Config 组件。

本章最后通过一个完整案例介绍针对微服务的各项实现技术及其具体的使用场景和方法。



## 第7章

# 微服务架构管理体系

上一章我们讨论了微服务架构的具体技术和工具,在本章中,我们将继续围绕如何实现微服务架构这一话题展开讨论。但本章关注于系统中各个微服务已经开发完成之后的阶段,即微服务架构的管理体系。

- 服务测试

服务测试基于传统的单元测试和集成策略方法,通过 Mock 和 Stub 消除服务与服务之间的依赖关系。但对于微服务架构而言,为了降低服务与服务之间进行集成和交互的成本,更为重要的是站在消费者的角度出发考虑测试的需求和实现策略,由此诞生了消费者驱动的契约测试。

- 服务部署

微服务架构能够流行的一大原因在于容器技术的持续发展,容器技术为服务化体系提供了部署和运维相关的基础设施。目前最流行的容器技术是 Docker,我们可以使用 Docker Compose 对微服务进行编排操作。同时,为了更好地管理服务交付,配置管理和持续集成相关的设计理念和工具也是整个服务体系构建过程中的重要组成部分。

- 服务监控

在本章中,服务监控包括两方面:一方面是日志聚合,重点介绍目前主流的 ELK 架构体系;另一方面是服务跟踪,服务跟踪的原理比较简单,在实现上 Spring Cloud 也为我们提供了专门的服务跟踪组件 Spring Cloud Sleuth。

- 服务安全

在分布式系统中,加密、授权、认证以及以 HTTPS、OAuth 为代表的安全性协议是确保系统安全性的基本手段,而对于微服务架构,需要考虑外部应用接入、用户到服务的鉴权、服务到服务的鉴权等多种鉴权场景,以 JWT 为代表的客户端 Token 技术得到了广泛应用。当然, Spring Cloud 中也提供了 Spring Cloud Security 作为微服务架构的安全性实现工具。

## 7.1 服务测试

对于软件中的任何功能,我们都需要进行测试。测试是一门综合性的技术,很多测试的理



念和方法具有通用性，但本书不打算对这些测试的基本概念和实现方式做过多的介绍，而是专注于微服务测试这一角度，探讨在微服务架构中开展测试工作与其他功能性测试或非功能性测试在操作过程中的不同点，并给出相应的工程实践。

### 7.1.1 微服务测试的维度

针对微服务测试，我们首先需要明确以下两个问题。本节内容将围绕这两个问题做具体展开。

- 我们需要测试什么？
- 我们应该采用什么样的测试方法？

#### 1. 测试内容

微服务的内部结构和外部集成方式决定着测试的内容。一方面，服务内部的功能需要测试；另一方面，服务与服务之间可以相互集成，而服务与外部环境之间也可以通过交互完成复杂的业务功能，这部分内容也是测试的重点。图 7-1 给出了一个典型的微服务在结构上的各个组成部分以及与外部的交互方式。

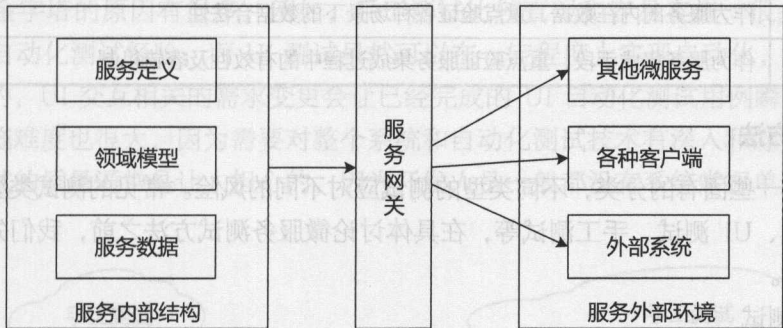


图 7-1 微服务的内部结构和外部交互方式

基于图 7-1 并从测试角度出发，对于一个微服务我们可以得出以下测试内容。

- 服务定义

服务定义往往涉及服务请求和返回的数据格式以及服务所能够支持的各种操作语义，服务请求者和消费者所应该遵循的交互方式和协议也属于服务定义的元数据。当然，我们可以根据针对不同的服务规定它应该遵循的响应时间、访问并发量支持等约束。

- 领域模型

领域模型代表服务背后的业务逻辑，我们在 2.3 节中已经阐述了子域、界限上下文等面向领域的策略设计相关内容。领域驱动设计包含了如何建立领域模型的完整体系和方法论，本书无意对领域驱动设计做过多介绍。除了策略设计之外，关于服务内部所涉及的实体



（Entity）、值对象（Value Object）、领域服务（Domain Service）等面向领域的技术设计内容可参考相关资料做进一步了解和掌握<sup>[3]</sup>。

- 服务数据

面向业务的服务内部肯定需要保存和处理数据。服务数据指的就是服务内部所使用的数据模型以及存储方式，包括传统的关系型数据库，也包括各种 Nosql 技术以及以 Elastic Search 为代表的垂直化搜索引擎。

- 服务网关

服务网关负责服务与外部环境之间的交互和协作，一方面通过自身暴露服务，另一方面也需要从外部获取数据。服务网关一般不保存和处理业务相关数据，通常是实现系统集成。

表 7-1 进一步梳理了各项测试内容的测试重点。

表 7-1 微服务测试内容重点

测试内容	测试重点
服务定义	作为服务的对外表现形式，重点验证服务定义的统一性、合理性和稳定性
领域模型	作为服务业务逻辑的载体，重点验证各种业务场景下的功能性需求是否得到满足
服务数据	作为服务的内在数据，重点验证各种场景下的数据合法性
服务网关	作为服务集成手段，重点验证服务集成过程中的有效性及容错机制

## 2. 测试方法

测试存在一些固有的分类，不同类型的测试应对不同的风险。常见的测试类型包括单元测试、集成测试、UI 测试、手工测试等，在具体讨论微服务测试方法之前，我们先来回顾一下这些测试类型。

- 单元测试

单元测试（Unit Test）应由开发人员实现和使用，测试的对象是小段代码，目的是帮助开发人员确定源代码做了希望它做的事。同时，我们还应该注意到一个单元测试用例只能针对一个类，而且一个类的单元测试必须是完全独立的，不应再和其他代码有任何交互。这也是我们要使用 Mockito、Easymock 等 mock 框架的原因，当测试过程中某个对象必须和其他对象交互时，就使用 mock 对象来保持单元测试的独立性。单元测试可以是自动的，也可以是手动的。通常，我们会在系统的开发阶段手动执行单元测试，但开发单元测试的一个重要目的是为了在后续的功能回归中自动化运行，从而验证回归测试的效果。

- 集成测试

集成测试（Integration Test）旨在测试各个组件间是否能互相配合并正常工作。和单元测试一样，集成测试也是为了确认代码是否按设计或期望的方式工作。相较单元测试，集成测



试的范围比较宽泛。一方面，范围可以很小，例如，在一个测试用例中涉及了多个类，就可以认为这是集成测试；另一方面，范围也可以大到整个系统，包括从后台到前端的所有组件。显然，集成测试往往会涉及外部组件，如数据库、硬件、网络等。

### • 端到端测试

端到端测试（End to End Testing），也就是通常所说的系统测试（System Testing），是从用户使用系统的角度出发，对系统的行为进行正确性验证的测试。端到端测试包含所有访问点的功能测试及性能测试。很多时候，我们会认为端到端测试是一种黑盒测试，但从某种意义上来说，端到端测试实质上可以理解为一种“灰盒”测试，一种集合了白盒测试和黑盒测试长处的测试方法。

## 3. 测试金字塔

在一个软件系统中，所有的测试方法所占有的比例应该是不一样的，图 7-2 中的测试金字塔展示了各种测试方法的一种分布情况。可以看到，在所有的测试方法中，单元测试应该占有最大比例，其次是集成测试，而 UI 测试和手工测试的所占比例应该最低。

图 7-2 所示的实际上是一种理想状态，更常见的场景是测试倒金字塔（见图 7-3）。形成测试倒金字塔的原因有很多，例如，手工测试比较直观和容易实施，很多测试人员也没有足够的自动化测试经验；而 UI 测试虽然可以在一定程度上实现自动化，但 UI 测试通常都是脆弱的，UI 交互相关的需求变更会让已经完成的 UI 自动化测试用例瞬间不可运作；集成测试实施难度也很大，因为需要对整个系统和自动化测试技术有深入和综合的了解；最后，单元测试的质量通常是让人担心的，因为开发人员一般都没有系统掌握单元测试的实施方法。



图 7-2 理想状态下的测试金字塔

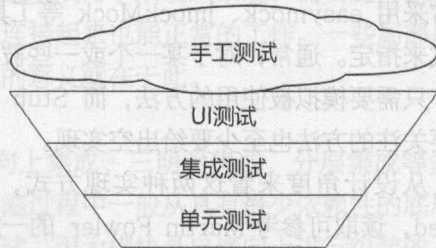


图 7-3 现实中的测试倒金字塔

测试金字塔给我们的启示是要开展自动化工作。我们将在 7.2 节服务交付与部署中介绍微服务与持续交付相关内容。



## 7.1.2 微服务测试实现方法

对于开发人员而言，单元测试和集成测试是最基本也是最有效的两类测试，本节结合微服务的特点对这两类测试进行展开。

### 1. 单元测试

单元测试通常是简单的，因为它的影响范围只在一个类之内，我们只需要根据业务逻辑编写相应的测试用例即可。但是当这个类依赖其他类或组件时，事情就变得没有那么简单。

#### (1) Mock 和 Stub

在介绍如何在单元测试中处理一个类与其他类或组件之间的依赖关系之前，我们先来明确两个在测试领域比较容易混淆的概念，即 Mock（模拟）和 Stub（打桩）。

首先，非常明确的一点，Mock 和 Stub 都可以用来对系统（或者将粒度放小为类和组件）进行隔离。在单元测试中，我们通常关注的是测试对象的功能和行为，对于测试对象涉及的一些依赖，我们仅仅关注他们与测试对象之间的交互。例如，是否调用、何时调用、调用的参数、调用的次数和顺序，以及返回的结果或发生的异常等。至于这些被依赖对象如何执行这次调用的具体细节，通常我们并不关注。因此，常见的技巧就是用 Mock 对象或者 Stub 对象来替代真实的次要对象，模拟真实场景来开展对测试对象的测试工作。

然而，Mock 和 Stub 的区别也非常明显，从类的实现方式上看，Stub 有一个显式的类实现，按照 Stub 类的复用层次可以实现为普通类（被多个测试用例复用）、内部类（被同一个测试用例的多个测试方法复用）乃至内部匿名类（只用于当前测试方法）。对于 Stub 的方法也会有具体的实现，哪怕简单到只有一个简单的 return 语句。而 Mock 则不同，Mock 的实现类通常采用 easymock、jmockMock 等工具包来隐式实现，具体 Mock 方法的行为则通过模拟方式来指定。通常，对于某一个或一些被测试对象所依赖的测试方法而言，Mock 编写相对简单，只需要模拟被使用的方法，而 Stub 要复杂一些，需要实现这个类的所有逻辑，即使是不需要关注的方法也至少要给出空实现。

从设计角度来看这两种实现方式，需要引入两个概念：interaction-based 和 state-based，读取可参考 Martin Fowler 的一篇文章“Mocks Aren’t Stubs”<sup>[20]</sup>来获取对这两个概念的权威解释。我们这里只给出结论，Stub 是 state-based，关注的是输入和输出，而 Mock 是 interaction-based，关注的是交互过程。

#### (2) 单元测试实施策略

Mock 和 Stub 在单元测试中应用非常广泛，这点对微服务而言同样适用。图 7-4 描绘了在微服务中使用 Mock 和 Stub 的示意图。在现实开发过程中，我们当然也可以使用真实的类或组件的实现来消除测试对象对他们的依赖，所以在图 7-4 中也添加了这种实现方式。



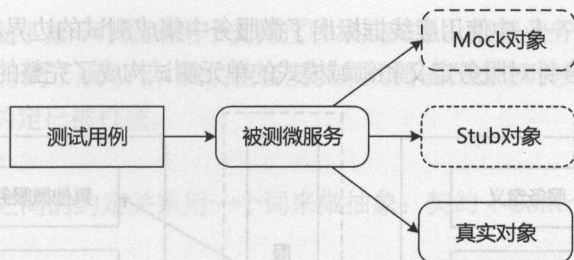


图 7-4 单元测试与依赖关系

在形式上，Mock 是在测试代码中直接 Mock 类和定义 Mock 方法的行为，测试代码和 Mock 代码通常是放在一起的，因此测试代码的逻辑也容易从测试用例的代码上体现出来。而对于 Stub 而言，如测试逻辑复杂，Stub 数量就会很多并且某些 Stub 需要传入一些如 true、false 之类的标记来设定不同的行为，测试逻辑的可读性就会下降。

另外，Mock 通常很少考虑复用，每个 mock 对象都是遵循“just enough”原则，一般只适用于当前测试方法。而 Stub 则通常比较方便复用，尤其是一些通用的 Stub。所以，Mock 的优点是构建成本低但维护成本较高，而 Stub 则刚好相反。具体采用何种策略需要追求一定的平衡性。

### （3）单元测试的内容

在测试内容上，我们认为服务定义和领域模型适合开展单元测试。而服务数据和服务网关由于一般都会涉及数据库、外部资源、其他服务等依赖，比较适合采用的测试方式是集成测试。

## 2. 集成测试

在单元测试的基础上，将所有模块按照设计要求组装成为子系统或系统，需要进行集成测试。一些类和组件虽然能够单独工作，但并不能保证连接起来也能正常的工作。一些局部反映不出来的问题，在全局上很可能暴露出来，集成测试的意义就在于此。

### （1）集成测试的策略

集成测试在开展策略上有自顶向下集成、自底向上集成、三明治集成、分层集成等多种具体方式。以最常使用的自底向上集成测试为例，实施过程中一般从具有最少依赖性的底层模块开始，按照由底向上的顺序构造系统并进行集成测试。因为模块是自底向上进行组装的，对于一个给定层次的模块，它的子模块以及子模块的所有下属模块事前都已经完成组装并经过测试，所以可以采用这种迭代和累加式策略完成整个系统的测试。

### （2）集成测试的内容

对于微服务而言，集成测试关注点一方面在于服务与外部资源的交互行为是否正确，这里的外部资源包括其他的微服务，也包括外部的第三方服务；另一方面也需要确保服务内部数据的正确性，这里的数据来源一般有关系型数据库、Nosql 或垂直化搜索引擎。



基于以上分析,图 7-5 中使用虚线框标出了微服务中集成测试的边界。在服务数据和服务网关上开展集成测试,结合针对服务定义和领域模式的单元测试构成了完整的微服务测试范围。

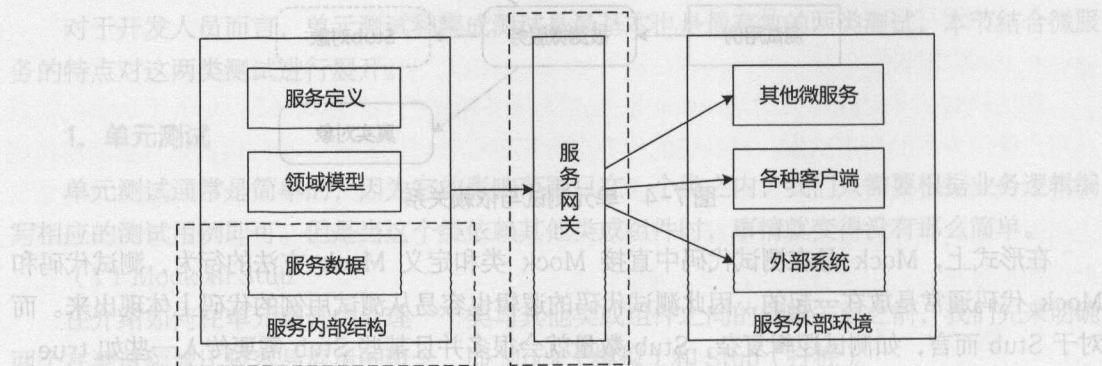


图 7-5 集成测试的边界

### 7.1.3 消费者驱动的契约测试

对于任何一个类或组件所暴露的对外接口,我们都可以把他们归为一种契约(Contract):接口的调用者希望通过接口获取某种约定的价值,然而很多时候这种契约并没有被正式地约定。

#### 1. 消费者驱动契约测试的理念

在微服务中,当服务没有满足约定,错误就会产生在生产环境或集成测试环节。前面介绍的集成测试虽然能够发现并解决部分因为违反接口约定所带来的错误,但集成测试本身也会存在一些问题。最典型的场景就在于随着服务的不断迭代,接口也会相应地产生变化。如果服务数量较多,这种变化会导致集成测试结果的不稳定。图 7-6 展示了由于接口变化导致集成测试失败的一种场景。

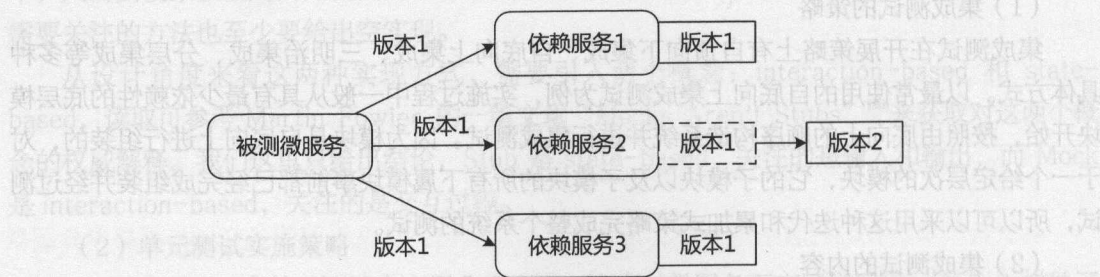


图 7-6 接口版本变化导致集成测试失败的场景

在图 7-6 中,被测微服务依赖服务 1、服务 2 和服务 3 等三个不同的服务,在现有情况



下，这三个服务都开发了第一个版本用于支持这种集成关系。根据业务需要，服务 2 做了一次服务升级，从版本 1 升到了版本 2，我们就会发现集成测试在这个时候会发生错误，这种错误就来自于接口的既有约定已被打破。

(1) 契约的概念

我们可以把接口之间的约定关系用一个词来做抽象：契约（Contract）。契约在组成上有以下几个方面。

- 数据格式  
定义其他微服务所期望的数据格式，以及如何将这些数据传递给对方。
- 接口定义  
接口所能提供的可用操作。
- 访问协议  
能够通过何种协议或步骤完成接口定义中的具体操作。
- 非功能性描述  
调用时延或吞吐量等非功能性约束和条件。

对于服务的提供者和消费者而言，存在不同的契约表现形式。服务提供者契约包含了服务提供者所能提供的所有内容。一个服务提供者仅包含一个这种契约，而且这种契约一般会随着版本的演进而不断变化，正如图 7-6 中服务 2 所示的效果一样。

消费者契约则包含对服务进行利用的各个方面，一个服务可以存在一个或多个消费者契约。这种契约只包含某个或某些消费者真正使用的部分服务定义，并且根据服务消费者的变更而做相应的调整。图 7-7 展示了服务提供者契约和消费者契约之间的这层关系。

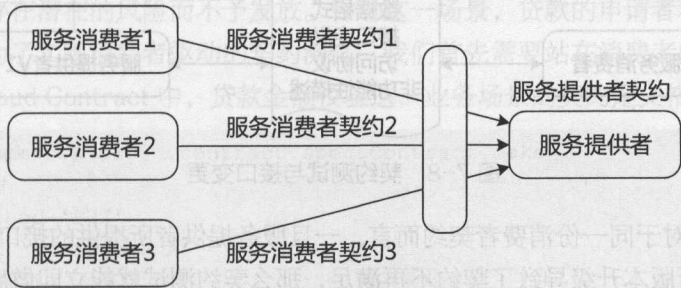


图 7-7 服务提供者契约和消费者契约的区别

(2) 消费者驱动的契约测试

所谓消费者驱动契约（Consumer Driven Contract, CDC）就是从消费者的角度出发驱动消费者协议的设定和调整。消费者驱动的契约描述的是服务提供者向当前所有服务消费者承诺遵守的约束。各消费者通过创建消费者驱动的契约，把自己的具体期望告知提供者。如果服务提供者



接受了一个消费者驱动的契约,那么它只需保证已有约束仍能得到满足,即可自行修改和发布。

消费者驱动的契约测试是针对微服务接口进行的测试,它能验证服务提供者所提供的契约是否满足消费者的期望。对于一个服务提供者而言,每个消费者会根据与其交互场景和上下文的不同产生不同的契约。当这个服务提供者频繁变更时,就应该保证每个消费者依然能够正确地消费契约。

消费者驱动的契约测试能够提供一定机制验证提供者所提供的服务能否始终满足契约。显然,对于微服务而言,我们只需要关注服务所暴露出来的接口 API 即可。因为每个消费者拥有自身的消费者契约,只需要根据消费者契约编写独立的测试用例验证服务提供方所暴露出来的那一部分接口即可。这些测试用例仅仅关注契约是否满足期望,而不需要深入测试服务内部的行为,所以开展方式相较集成测试具有轻量级的优点,可以在很大程度上降低测试成本。

消费者驱动的契约测试能够帮助服务消费者和提供者验证在协作过程中接口是否已经发生变化。每当服务提供者所暴露的接口发生变更,契约测试就能检测到该接口是否仍然和契约所要求的保持一致,图 7-8 展示了这一过程。

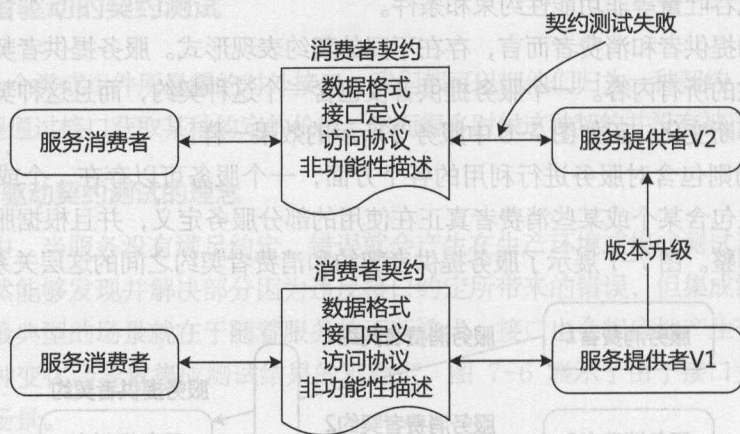


图 7-8 契约测试与接口变更

图 7-8 中,对于同一份消费者契约而言,一旦服务提供者所提供的接口从版本 1 上升到版本 2 时,如果新版本升级导致了契约不再满足,那么契约测试就能立即做出验证,从而在开展功能测试之前就能尽早发现错误。

## 2. 消费者驱动契约测试的系统方法

消费者驱动契约测试的系统方法包括设计合理的测试过程以及采用合适的测试工具。

### (1) 消费者驱动契约测试过程

消费者驱动契约测试的目的是为了明确契约的有效性。基于图 7-8 中的思路,消费者驱



动契约测试可以采用以下步骤验证契约是否已经发生变更。

- 业务场景提取

根据业务需求,选择合适的测试场景。并不是所有的业务场景都需要使用消费者驱动契约测试,往往越容易发生变更的业务场景就越需要进行测试。针对移动医疗系统,预约挂号的号源信息、医生排班信息等都是比较合适的测试场景。

- 消费者请求契约化

消费者发送的请求、提供者提供的响应以及关于业务场景的元数据都需要明确记录,并整理成该场景下的契约。

- 模拟消费者发送请求

模拟消费者,向真实的服务提供者发送请求。

- 提供者验证

通过获取请求结果,验证提供者提供的契约是否已经发生变化。

## (2) 消费者驱动契约测试工具

目前,业界支持消费者驱动契约测试工具常见的有 Pact (<https://github.com/realestate-com-au/pact>)、Pacto (<https://github.com/thoughtworks/pacto>) and Janus (<https://github.com/gga/janus>)。当然,作为一个完整的微服务套件,Spring Cloud 也提供了 Spring Cloud Contract 作为消费者驱动契约测试的开发框架。这里以 Spring Cloud Contract 为例对消费者驱动契约测试工具的使用方式做简要介绍,我们通过对官方示例的解读来理解 Spring Cloud Contract 的使用方法。

考虑向银行申请贷款的业务场景,我们需要考虑贷款人的还款能力,对那些金额明显过高的申请应该认为存在潜在的风险而不予发放。针对这一场景,贷款的申请者和发放者显然分别是两个微服务,为了实现消费者驱动的契约测试,我们首先需要站在消费者的角度出发定义契约。在 Spring Cloud Contract 中,贷款金额校验这一业务场景的契约定义格式如下。

```
org.springframework.cloud.contract.spec.Contract.make {  
    request {  
        method 'PUT'  
        url '/fraudcheck'  
        body([  
            "client.id": $(regex('[0-9]{10}')),  
            loanAmount: 99999  
        ])  
        headers {  
            contentType('application/json')  
        }  
    }  
    response {
```



```
status 200
body([
    fraudCheckStatus: "FRAUD",
    "rejection.reason": "Amount too high"
])
headers {
    contentType('application/json')
}
}
```

在以上契约中，我们可以看到基于 HTTP 协议的请求和响应结果，包括所使用的 HTTP 方法、协议头和协议体内容。

站在契约的两端，消费者作为服务访问的客户端可以使用 RestTemplate 进行 HTTP 访问，示例代码如下。

```
ResponseEntity<FraudServiceResponse> response =
    restTemplate.exchange(
        "http://localhost:" + port + "/fraudcheck",
        HttpMethod.PUT,
        new HttpEntity<>(request, httpHeaders),
        FraudServiceResponse.class);
```

而在服务器端，为了验证服务调用的正确性，我们需要开发与契约对应的 Stub，在 Spring Cloud Contract 中，创建 Stub 的过程可以做到自动化。只要在测试用例类上添加 @AutoConfigureStubRunner 注解并运行测试用例就能做到这一点。系统自动生成的 Stub 代码如下。有了 Stub 代码也就有了满足契约的服务输入和输出，接下去的事情就是根据测试用例完成业务代码即可。

```
@Test
public void validate_shouldMarkClientAsFraud() throws Exception {
    MockMvcRequestSpecification request = given()
        .header("Content-Type", "application/vnd.fraud.v1+json")
        .body("{\"client.id\":\"1234567890\",\"loanAmount\":\"99999\"}");

    ResponseOptions response = given().spec(request)
        .put("/fraudcheck");

    assertThat(response.statusCode()).isEqualTo(200);
    assertThat(response.header("Content-Type"))
        .matches("application/vnd.fraud.v1.json.*");

    DocumentContext parsedJson = JsonPath.parse(response.getBody().asString());
    assertThatJson(parsedJson).field("[ 'fraudCheckStatus' ]")
        .matches("[A-Z]{5}");
    assertThatJson(parsedJson).field("[ 'rejection.reason' ]")
        .isEqualTo("Amount too high");
}
```

关于 Spring Cloud Contract 的更多信息可以参考其官方网站 (<https://cloud.spring.io/spring->



cloud-contract/single/spring-cloud-contract.html)。

## 7.2 服务交付与部署

微服务架构中，服务数量众多且由不同的团队独立维护和管理，这给服务交付和部署带来了挑战，因为我们将不得不面对如何进行服务版本控制、如何建立统一发布过程、如何开展团队之间的有效协作等问题。针对这些问题，一方面我们可以借助于传统的配置管理和持续集成手段来对微服务进行交付管理；另一方面，以 Docker 为代表的容器技术的发展为微服务的兴起提供了服务部署上的支持，在本节中，我们也将介绍基于 Docker 的微服务部署体系和相关实现方案。

### 7.2.1 微服务交付管理

当我们完成微服务代码的开发，从代码提交开始到最后的发布，不同的团队可能会有很多步骤。比较典型的软件交付工作流程可以抽象为代码提交→代码编译→自动化测试→手工测试→部署→预发布→正式发布，这个过程中每一步都可能出错。

针对软件交付基本步骤中的出错情况，我们做一下根源分析，会发现大多数错误的发生原因一方面是因为没有尽早发现问题，另一方面则来源于手工配置和手工部署。通常，系统上线的第一次发布因为周期很长，各个团队之间没有形成体系化工作规范所造成的协作成本和出错概率会比较高，导致很多潜在的问题并没有在发布流程中较早地暴露出来。有时候，我们也会提出类似的抱怨“为什么试运行环境正确，生产环境却有问题”。在复杂环境下，配置管理混乱、缺少必要的自动化等因素会导致手工操作的结果不可控。

基于以上分析，软件交付基本思路就是要做到自动化发布。自动化代表可重复，并注重过程，过程对则结果一定对。在自动化发布环境下，无论什么修改都应该触发相应的监控流程，确保问题在第一时间被暴露和解决。一旦建立起自动化发布平台，就可以通过频繁发布降低出错所引起的风险，而频繁发布也能够促进快速反馈，从而推动过程改进。

本节将围绕微服务交付模型和需求进行讨论，我们的基本思路是关注自动化与集成，并基于原则导出工程实践。要想实现自动化，首先要从配置管理入手，然后借助于持续集成的相关理念和模式。

#### 1. 配置管理

随着微服务数量的不断增加，对配置管理（Configuration Management，CM）的需求越来越多。软件配置由在软件工程过程中产生的所有信息项构成，它可以看作该软件的具体形态



在某一时刻的瞬间影像，而软件配置管理（Software Configuration Management，SCM）就是对这些形态和影像的管理过程。

一方面，协调软件开发从而使混乱减到最小是软件配置管理的目标，采用对软件的修改进行标识、组织和控制的技术使错误量降至最低，并使生产率最高。另一方面，配置管理能够把握系统的变更处理，从而使软件系统可以随时保持其完整性，可以用来评估和跟踪提出的变更请求，并保存系统在不同时间的状态。

配置管理中存在一批基本元素，在具体介绍配置管理的具体活动之前，我们有必要了解这些基本元素（见表 7-2），并对其中几个重要元素进行展开。

表 7-2 配置管理基本元素

元 素	描 述
软件配置项（SCI）	与配置控制下的软件项目有关的任何事物
版本（Version）	配置项的一个实例
基线（Baseline）	组成系统的各个组件版本的集合，不能改变
配置管理数据库（CMDB）	保存配置项及其关系的数据库
主线（Mainline）	代表系统不同版本的基线的序列
发布版本	发布给客户使用的系统版本
工作区间（Workspace）	一个私有的工作区间，在其中可以修改而不至于影响其他会修改软件的开发者
分支（Branch）	从现存的代码线版本中构建一个新的代码线
系统构建	通过耦合和链接组件和库的合适版本创建一个可执行的系统版本

#### • 软件配置项

计算机程序（源代码和可执行程序）、描述计算机程序的文档（针对技术开发者和用户）、数据（包含在程序内部或外部）等项目包含了所有在软件过程中产生的信息，总称为软件配置项。软件配置项具有名称、描述、类型（模型元素、程序、数据、文档等）、项目标识符、变更和版本信息等属性。

#### • 基线

所谓基线是指已经通过正式评审和批准的软件规格说明或代码，它可以作为进一步开发的基础，并且只有通过正式的变更规程才能进行修改。在软件配置项成为基线之前，可以迅速地变更；一旦成为基线，变更时就需要遵循正式的评审流程才可以变更。因此，基线可看作是软件开发过程中的里程碑。围绕基线所开展的工作流程可参考图 7-9。



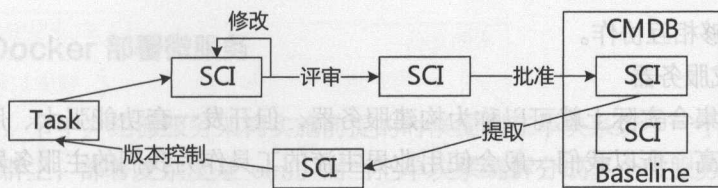


图 7-9 基线流程

- CMDB

图 7-9 中的 CMDB 是指配置管理数据库 (Configuration Management Database)，是用于保存与软件相关的所有配置项的信息以及配置项之间关系的数据库，包括每个配置项及其版本号、变更可能会影响到的配置项、配置项的变更路线及轨迹、与配置项有关的变更内容以及不同配置项之间的关系等。

围绕配置管理的相关元素，我们可以总结出主线 (Mainline)、任务分支 (Task Branch) 等模式，由于篇幅有限，这里不做过多展开。关于配置管理的更多探讨可参考《系统架构设计》<sup>[1]</sup> 中的描述。

## 2. 持续集成

持续集成 (Continus Integration, CI) 强调尽早发现问题，降低缺陷进入下一环节的比率，从项目管理角度讲也意味着节约成本和降低风险，并提高了项目的可见性。对于开发团队而言，持续集成理念的引入可以帮助团队成员培养一种意识，建立团队对开发产品的信心。持续集成在构成上包括角色、版本库、构建脚本、持续集成服务器和反馈等元素。

- 多种角色

实现持续集成需要多种角色参与其中，也能切实解决这些角色所面临的一些问题。对于开发人员而言，重复工作太多导致工作效率降低是一个痛点；而测试人员经常抱怨旧 bug 还没解决又出很多新 bug；为了降低发布风险，运维人员经常需要半夜发布版本。通过持续集成就能够把开发人员的部分重复性工作自动化，控制和降低 bug 率并实现按需发布版本。开发、测试和运维也恰恰构成了 DevOps 中的三个维度，从这个角度讲，持续集成可以是 DevOps 的一种表现形式。

- 版本库

带有版本控制功能的中央仓库是实现持续集成的基础，关于版本库的工具和实践属于上文配置管理中的内容范畴。

- 构建脚本

自动化是我们的目标，实现自动化的基本手段就是通过各种构建脚本把原本需要手工执行的步骤转变为系统自动执行。通常，构建脚本的目的在于集成各种第三方工具并通过一定的策



略使这些工具能够相互协作。

- 持续集成服务器

构建脚本的集合实际上就可以称为构建服务器，但开发一套功能强大、用户体验好的集成服务器成本太高，所以我们一般会使用业界主流的工具作为我们的主服务器，这些持续集成服务器都提供了较高的可扩展性，可以通过编写部分构建脚本并嵌入其中实现集成的定制化需求。

- 反馈

反馈即通过一系列的监控机制确保集成过程中每一个步骤都能进行审查和确认，并提供邮件、IM 等一系列反馈机制，确保尽早发现问题并解决问题。这一点同样与软件交付模型的目标相一致。

引入持续集成的过程也是一个循序渐进的过程，在没有任何持续集成经验的团队中，尝试从新系统开始是一个不错的选择，对于遗留系统则倾向逐步过渡。无论新老系统，当项目开始时都该采取行动并开展多角色协作，最好有专人负责持续集成服务器中的追踪结果。图 7-10 是引入持续集成之后的工作流程，我们可以看到，从以 Redmine 为代表的问题跟踪系统开始，伴随着全局唯一的 IssueId，任何需求和 bug 都会流转到开发人员，开发人员通过版本控制系统进行个人工作区间和中央仓库之间的交互，最终所有的版本共享通过持续集成服务器生成可交付成果。

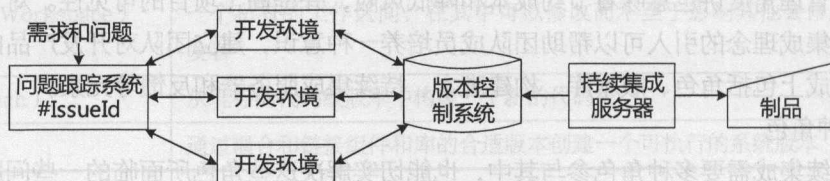


图 7-10 持续集成流程图

图 7-10 中的持续集成服务器是一套完整的解决方案，能够从持续测试、持续审查、持续部署和持续反馈等维度提供支持。

持续测试包括自动化单元测试、自动化组件测试、自动化系统测试以及各种测试分类和可重复特性。持续审查通过采用组织级别的标准建立代码审查机制并持续进行设计检查，试图从减少重复代码、判断代码覆盖率、降低代码复杂度等角度为开发人员提供代码质量的可视化视图。持续部署要求随时随地发布可工作的软件，通过持续集成服务器，能够为每一个构建打上标签、执行所有测试、构建反馈报表，并具备回滚构建的过程能力。持续反馈提倡以正确的方式在正确的时间将正确的信息传递给正确的人。这里正确的方式可以是电子邮件、声音和 IM；正确的时间应该支持每天、每周或问题发生时；正确的信息包括构建状态、审查报表、测试结果；而正确的人通常包括系统构建人员、开发人员以及架构师。



## 7.2.2 基于 Docker 部署微服务

我们在 1.6.1 节中介绍微服务架构实施前提的时候提到在单块系统中一个应用部署到一台主机或在主机集群上，部署复杂度是  $O(n)$ 。而当把单块系统拆分成多个微服务之后，其部署复杂度就上升到  $O(n^3)$ ，由此引出的一个话题就是基础设施的自动化。自动化的部署方案能够提高工作效率、降低风险，而目前基于容器的部署机制在业界得到了广泛应用。

### 1. Docker 简介

Docker 作为容器为微服务部署和实施带来了如下转变。

- 轻量级的部署建模方式

Docker 所提供的部署建模方式是轻量级的，也就是说我们在没有使用过 Docker 的前提下把部署向 Docker 化转变是一个比较简单的过程。

- 开发和部署职责逻辑分离

通过 Docker 实现的容器管理机制，开发人员关注应用程序，运维人员关注管理容器，从而更好地实现开发和部署的职责分离。

- 快速/高效的开发生命周期

容器技术以及围绕容器技术开展的基础设施自动化建设能够提升服务部署和运维的效率，缩短上线时间，同时也更加有利于服务的构建和团队的协作。

- 适合微服务的架构体系

微服务的一大特征是独立部署和独立微服务，容器技术恰好就提供了单个容器运行单个应用程序的部署方式，非常适合于微服务架构。

### 2. Docker 组件与命令

关于 Docker 的讨论，我们将从两个方面进行切入，一方面介绍 Docker 的核心组件，另一方面则是关注于如何使用 Docker，也就是 Docker 提供的各项命令。

#### (1) Docker 组件

Docker 架构图见图 7-11，我们可以看到图中包含了客户端、主机和注册中心三个层次，这些层次涵盖 Docker 运行时所涉及的客户端、守护进程、镜像、容器等各个核心组件。

- 守护进程

守护进程 (Daemon) 是运作在 Docker 宿主 (Host) 的后台程序，负责响应来自客户端的请求，也就是各种命令。

- 客户端

客户端 (Client) 与 Daemon 通信，提交命令。



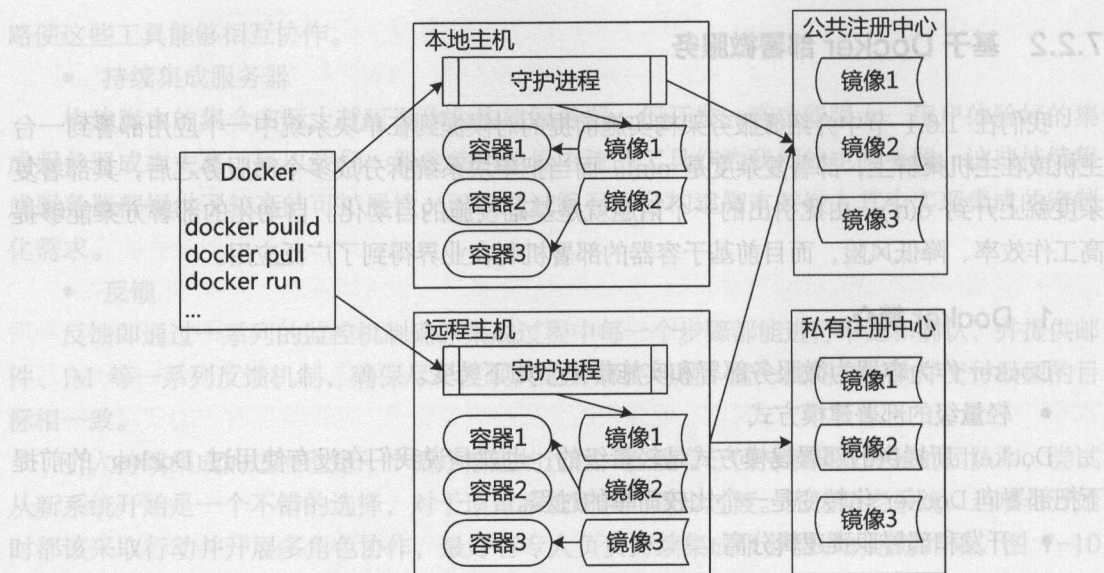


图 7-11 Docker 架构图

- 镜像

镜像（Image）相当于源代码的构建环境，由一系列指令组成。

- 容器

容器（Container）则是源代码的运行环境。

- 注册中心

注册中心（Registry）保存已经构建完成的镜像。

我们可以在图 7-11 中看到 Docker 主机分为本地主机和远程主机两类，每个主机都运行着一个守护进程。同时每个主机内部也可能存储数量不一的镜像和容器，其中守护进程指向镜像，镜像再指向容器。注册中心分为公共与私有两种，其中公共注册中心又被称为 Docker Hub，存放着数量庞大的公共镜像，Java EE 领域中的常见工具（如 JDK、Tomcat、Mysql 等）都有公共镜像；而私有注册中心则面向内部团队，包含着团队所特有的镜像文件。关于公共与私有注册中心的含义，可以类比 Maven 的私有服务器和公共服务器的概念。

## （2）Docker 命令

Docker 的常用命令有以下几种。

- info

查看 Docker 工作状态。

- run

拉取镜像，运行容器。该命令可以带有一些参数，如 `-name` 表示命名容器、`-d` 表示运行



的是一个守护式容器、-p 表示映射端口。

- start/stop  
启动/停止容器。

- images  
查看镜像列表。

- build  
使用 Dockerfile 构建镜像。

- history  
查看镜像构建过程。

### 3. Dockerfile

前面提到在 Docker 命令里有一项 build 命令使用 Dockerfile 构建镜像，而 Docker 命令正是通过 Dockerfile 形成一组命令集合以完成镜像的构建。常见的 Dockerfile 命令如下。

- FROM

命令格式为 FROM <image>或 FROM <image>:<tag>，该命令指定所使用的基础镜像，例如，FROM java:8 代表基于 JDK8 开始构建我们自己的镜像。

- MAINTAINER

命令格式为 MAINTAINER <name>，指定维护者信息。

- RUN

命令格式为 RUN <command>或 RUN ["executable", "param1", "param2"]。该命令的意思是在当前镜像基础上执行指定命令，并提交为新的镜像。

- CMD

命令格式为 CMD ["executable", "param1", "param2"]，通过该命令，我们能指定启动容器时执行的命令。因为跟容器启动生命周期有关，所以每个 Dockerfile 只能有一条该种命令。

- EXPOSE

命令格式为 EXPOSE <port> [<port>...]，该命令告诉 Docker 服务端容器暴露的端口号，供系统交互时使用。

- ENV

命令格式为 ENV <key> <value>，该命令用于指定一个环境变量，会被前面的 RUN 命令所使用。

- ENTRYPOINT

命令格式为 ENTRYPOINT ["executable", "param1", "param2"]，代表配置容器启动后



执行的命令。同样因为跟容器生命周期有关，每个 Dockerfile 只能有一条该种命令。

- ADD

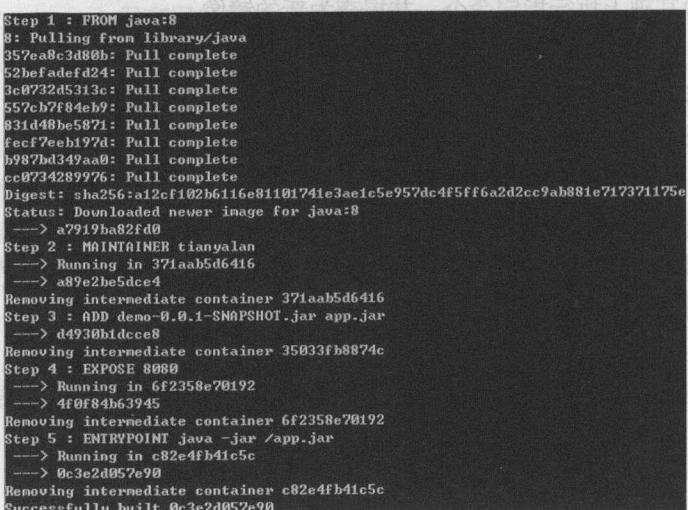
命令格式为 ADD <src> <dest>，代表复制指定的 <src> 到容器中的 <dest>，我们往镜像里添加内容时就需要借助于这条命令。

下面给出一个最简单的 Dockerfile 示例，包含如下指令：

```
FROM java:8
MAINTAINER tianyalan
ADD demo-0.0.1-SNAPSHOT.jar app.jar
EXPOSE 8080
ENTRYPOINT ["java", "-jar", "/app.jar"]
```

以上命令的含义是指我们以 JDK8 为基础镜像开始构建我们自己的镜像，指定该镜像的维护者是 tianyalan，接着往镜像中添加了一个名为 demo-0.0.1-SNAPSHOT.jar 的 jar 包，并重新命名为 app.jar，然后暴露服务端为 8080。最后，我们通过 ENTRYPOINT 命令在容器启动成功之后运行 java -jar /app.jar 命令，可以想象这个命令是用来启动 app.jar。我们是通过 Spring Boot 技术构建这个 app.jar，所以可以直接使用 java -jar 来启动 jar 包中内置的 tomcat 服务器，从而启动服务。

上述 Dockerfile 示例所执行的过程效果参考图 7-12，可以看到首先会从公共注册中心中拉取 JDK8，然后依次执行各条命令。这里值得注意的是，每一个命令执行完之后，都会生成一串类似编号的字符串，这个字符串上代表的就是一个新的镜像编号，因为每执行完一条命令，我们都会得到一个新的镜像文件。



```
Step 1: FROM java:8
8: Pulling from library/java
357ea8c3d80b: Pull complete
52befade4d24: Pull complete
3c0732d5313c: Pull complete
557cb7f84eb9: Pull complete
831d48be5871: Pull complete
fecf7eeb197d: Pull complete
b987bd349aa0: Pull complete
cc0734289976: Pull complete
Digest: sha256:a12cf102b6116e81101741e3ae1c5e957dc4f5ff6a2d2cc9ab881e717371175e
Status: Downloaded newer image for java:8
--> a7919ba82fd0
Step 2: MAINTAINER tianyalan
--> Running in 371aab5d6416
--> a89e2be5dce4
Removing intermediate container 371aab5d6416
Step 3: ADD demo-0.0.1-SNAPSHOT.jar app.jar
--> d4930b1dcce8
Removing intermediate container 35033fb8874c
Step 4: EXPOSE 8080
--> Running in 6f2358e70192
--> 4f0f84b63945
Removing intermediate container 6f2358e70192
Step 5: ENTRYPOINT java -jar /app.jar
--> Running in c82e4fb41c5c
--> 0c3e2d057e90
Removing intermediate container c82e4fb41c5c
Successfully built 0c3e2d057e90
```

图 7-12 Dockerfile 执行效果示例



## 4. Docker Compose

在 Docker 中运行微服务的方式主要有两种,一种就是前面介绍的使用 Docker File 构建 Docker 镜像,而我们接下去要介绍的就是第二种方式,即使用 Docker Compose 编排微服务。

Docker File 方式面向的是单个微服务和单个容器,可以让用户管理一个单独的应用容器。但服务数量的增加也就意味着容器数量的增多,逐渐增加的容器数量为容器部署、运行及管理带来了挑战。Docker Compose 面向多个微服务和多个容器,专门用于解决多个容器的部署问题并提高部署方案的可移植性。

Docker Compose 是用来定义和运行复杂应用的 Docker 工具,使用 Docker Compose,我们就可以允许用户在一个模板 (YAML 格式) 中定义一个多容器应用,然后使用一条命令来启动整个应用。

### (1) Docker Compose 架构

Docker Compose 的整体架构如图 7-13 所示,从图 7-13 中我们可以把 Docker Compose 抽象成比较典型的三层结构,包括工程 (Project)、服务 (Service) 以及容器 (Container)。

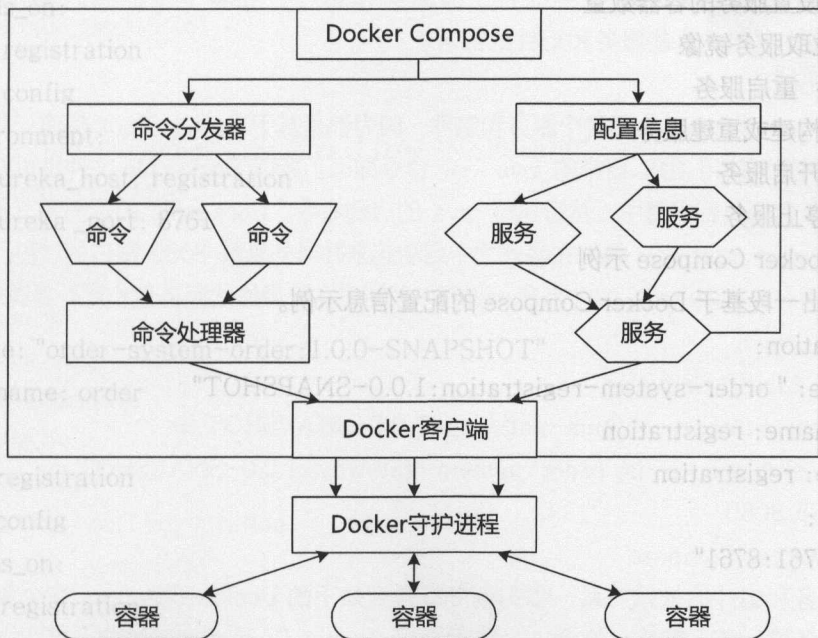


图 7-13 Docker Compose 架构图

#### • 工程

这里的工程与我们开发过程中的工程在概念上比较类似,一个工程涵盖了我们需要的所有



资源, 包括 Docker Compose 运行时所需的所有文件 (docker-compose.yml, extends 文件或环境变量文件等) 以及多个服务。

- 服务

这里的服务是一个抽象的概念, 每个服务中定义了容器运行的镜像、参数和依赖关系。一个服务中可包括多个容器实例。

- 容器

即指 Docker 容器。

### (2) Docker Compose 常见命令

Docker Compose 的常见命令罗列如下, 通过英文命名, 基本都可以明白具体的含义和执行效果。

up: 创建并启动容器

kill: 杀掉容器

ps: 显示容器

rm: 删除停止的容器

scale: 设置服务的容器数量

pull: 拉取服务镜像

restart: 重启服务

build: 构建或重建服务

start: 开启服务

stop: 停止服务

### (3) Docker Compose 示例

下面给出一段基于 Docker Compose 的配置信息示例。

registration:

```
image: "order-system-registration:1.0.0-SNAPSHOT"
hostname: registration
name: registration
ports:
  - "8761:8761"
```

config:

```
image: "order-system-config:1.0.0-SNAPSHOT"
hostname: config
name: config
```



```
links:
- registration

environment:
  eureka_host: registration
  eureka_port: 8761
product:
  image: "order-system-product:1.0.0-SNAPSHOT"
  hostname: product
  links:
  - registration
  - config
  depends_on:
  - registration
  - config
  environment:
    eureka_host: registration
    eureka_port: 8761
order:
  image: "order-system-order:1.0.0-SNAPSHOT"
  hostname: order
  links:
  - registration
  - config
  depends_on:
  - registration
  - config
  environment:
    eureka_host: registration
    eureka_port: 8761
```



我们从以上 Docker Compose 配置信息中不难看出整个系统的全貌。整个系统由四个服务构成，而且有两个基础服务，分别是 registration 和 config，可以猜想该系统应该是使用 Spring Cloud 框架进行开发的微服务系统。除了这两个基础服务之外，还有 product 和 order 这两个面向业务的微服务。

同时，我们还应该注意到有两个命令我们前面没有介绍，一个是 links 命令，另一个是 depends\_on 命令。depends\_on 命令用于指定服务依赖，一旦指定了服务依赖，将会优先于自身服务创建并启动依赖；而 links 命令用于链接另一容器服务，通过服务名进行服务识别，使用 links 命令同时也和 depends\_on 命令一样简单设置了服务依赖关系。在上文的例子中，config 服务依赖 registration 服务，通过 depends\_on 命令确保在服务启动顺序上就会先启动 registration 服务，然后再启动 config，这也符合 Spring Cloud 的服务部署要求。至于 product 和 order 则显然应该同时依赖 registration 和 config 服务。

## 5. 服务部署示例

在本节中，我们将对上一章 6.4.3 节中讨论的 Order-System 系统做进一步的扩展。在上一章中，我们已经构建了系统中的各个微服务和组件，在本节中，我们将介绍如何通过 Docker 相关技术完成这些服务和组件的部署。

### （1）Dockerfile

对于 Order-System 中的各个服务和组件，因为都是基于 Spring Boot 构建，所以启动方式非常简单，只需要使用 JDK 自带的 java -jar 命令即可。所以 Dockerfile 文件也可以非常简单，只需要将 jar 包拷贝到容器中，然后执行 java 的启动命令。Dockerfile 的示例如下所示，展示了把 order-system-gateway 组件在容器之中启动起来并对外暴露 8080 端口的过程。而对于其他服务或组件而言，Dockerfile 的内容是一致的，区别仅在于最终的服务包和对外暴露端口不同。

```
FROM java:8
MAINTAINER tianyalan
COPY target/order-system-gateway-1.0.0-SNAPSHOT.jar
CMD /usr/bin/java -jar order-system-gateway-1.0.0-SNAPSHOT.jar
EXPOSE 8080
```

### （2）Docker Compose

基于服务和组件的依赖关系，我们也可以得到如下的 Docker Compose 指令集合，对该系统中的微服务进行统一编排，这是上一节中介绍 Docker Compose 示例的完整版本，这些指令集合约定了各个服务之间的启动依赖关系。

```
registration:
  image: "order-system-registration:1.0.0-SNAPSHOT"
```



```
hostname: registration
name: registration
ports:
  - "8761:8761"
eureka_host: registration
config:
  eureka_port: 8761
  image: "order-system-config:1.0.0-SNAPSHOT"
  hostname: config
  name: config
  links:
    - registration
  depends_on:
    - registration
  environment:
    eureka_host: registration
    eureka_port: 8761
customer:
  image: "order-system-customer:1.0.0-SNAPSHOT"
  hostname: customer
  links:
    - registration
    - config
  depends_on:
    - registration
    - config
  environment:
    eureka_host: registration
    eureka_port: 8761
product:
  image: "order-system-product:1.0.0-SNAPSHOT"
  hostname: product
  links:
    - registration
    - config
    - gateway
  depends_on:
    - registration
    - config
    - gateway
  environment:
    eureka_host: registration
    eureka_port: 8761
```

## 7.3 服务监控

在微服务架构中，所有的业务逻辑都分布在不同的服务器上，如果服务运行出现错误，我们很难定位问题。在一台服务器上可以快速定位和处理问题。但在微服务架构中，显然事情就没有那么简单。微服务架构本质也是一种分布式架构，微服务架构的特点决定了各个服务



```

- registration
- config
depends_on:
- registration
- config
environment:
  eureka_host: registration
  eureka_port: 8761
order:
  image: "order-system-order:1.0.0-SNAPSHOT"
  hostname: order
  links:
    - registration
    - config
  depends_on:
    - registration
    - config
  environment:
    eureka_host: registration
    eureka_port: 8761
gateway:
  image: "order-system-gateway:1.0.0-SNAPSHOT"
  hostname: gateway
  links:
    - registration
    - config
    - customer
    - product
  depends_on:
    - registration
    - config

```

我们可以在 Docker Compose 配置信息中不难看出整个系统由 4 个容器组成，这 4 个服务构成，而其中 registration 和 config 是基础服务，分别是 registration 和 config，它们使用 Spring Boot 进行开发的微服务系统。除了这两个基础服务之外，还有 product 和 order 这两个服务。

同时，我们还需要注意到有两个命令我们前面没有介绍，一个是 links 命令，另一个是 depends\_on 命令。links 命令用于指定服务依赖，一旦指定了服务依赖，将会先启动被依赖的服务，然后再启动当前服务。这符合 Spring Cloud 的服务部署要求。至于 depends\_on 命令，它用于指定服务依赖，一旦指定了服务依赖，将会先启动被依赖的服务，然后再启动当前服务。这符合 Spring Cloud 的服务部署要求。至于 depends\_on 命令，它用于指定服务依赖，一旦指定了服务依赖，将会先启动被依赖的服务，然后再启动当前服务。这符合 Spring Cloud 的服务部署要求。

在本章中，我们构建了系统中的各个微服务和组件，在本章中，我们将介绍如何通过 Docker Compose 来部署这些服务和组件。

对于本章中的各个服务和组件，因为都是基于 Spring Boot 构建，所以启动方式非常简单，只需要将 jar 包放入容器中，然后运行 java 的启动命令。Dockerfile 的示例如下所示，展示了把 order-system-gateway 组件在容器之中启动起来并对外暴露 8080 端口的过程。而对于其他服务和组件而言，Dockerfile 的内容是一致的，区别仅在于最终的服务包和对外暴露端口不同。

基于本章的依赖关系，我们也可以得到如下的 Docker Compose 示例的完整版本，这些命令集合起来，就是本章中介绍的微服务系统的完整部署方案。



```
- customer
- product
- order
environment:
  eureka_host: registration
  eureka_port: 8761
ports:
  - "80:80"
monitor:
  image: "order-system-monitor:1.0.0-SNAPSHOT"
  hostname: monitor
  links:
    - registration
    - config
    - customer
    - product
    - order
    - gateway
depends_on:
  - registration
  - config
  - customer
  - product
  - order
environment:
  eureka_host: registration
  eureka_port: 8761
```

## 7.3 服务监控

我们知道在传统的单块系统中,所有的业务都在同一台服务器上,如果服务运行时出现错误和异常,我们只要关注一台服务器就可以快速定位和处理问题。但在微服务架构中,显然事情就没有那么简单。微服务架构本质也是一种分布式架构,微服务架构的特点决定了各个服务



部署在分布式环境中。各个微服务单独部署运行，彼此通过网络交互，而且都是无状态的服务，一个客户端请求可能需要经过很多个微服务的处理和传递才能完成业务逻辑。在这种场景下，我们难免会遇到这样的问题：

- 分散在各个服务器上的日志如何处理？
- 如果业务流程出现了错误和异常，如何简单快速定位问题？
- 如何跟踪业务流的处理顺序和结果？

我们发现，以前在单块系统下容易处理的日志监控，在微服务架构下却成了一个大问题。如果无法对日志进行聚合，我们将耗费大量时间来查找和定位问题，这也是为什么我们要把服务监控作为单独一节来讲的原因所在。在本节中，我们将从日志聚合和服务跟踪两个维度对服务监控这个主题展开讨论。

### 7.3.1 日志聚合

在分布式系统中，日志被分散储存在不同的服务器上。如果没有集中化日志环境，我们只能登录到具体某一台服务器进行日志查阅，服务器数量一多，效率就极其低下。另一方面，就算日志能够聚合到一起进行集中化管理，日志的统计和检索又成为一件比较麻烦的事情。如果没有友好、强大的可视化集成功能，对于要求更高的查询、排序和统计也无法得到令人满意的效果。

本节将要介绍的开源实时日志分析 ELK 平台能完美解决上述问题，它提供了目前业界关于日志聚合的代表性实现技术。

#### 1. ELK 基本架构

ELK 基本架构由 Elastic Search、Logstash 和 Kibana 三个开源工具组成。基于以上三个开源工具所构成的 ELK 整体架构如图 7-14 所示，接下来我们将对这三个工具做简要展开。

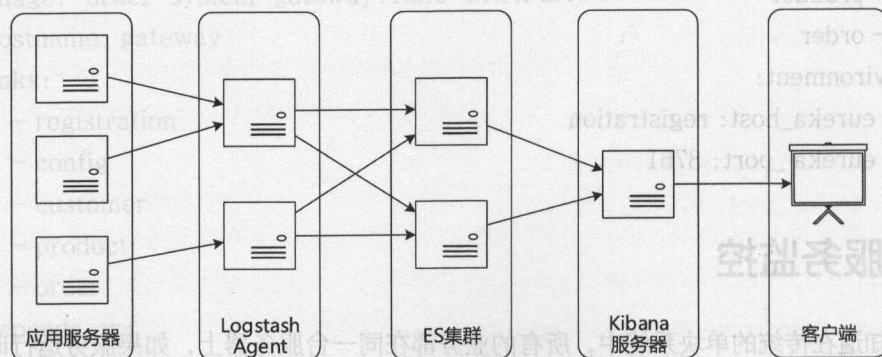


图 7-14 ELK 整体架构图



### (1) Elastic Search

Elastic Search 是一个实时的分布式搜索和分析引擎，它可以用于全文搜索、结构化搜索及分析。它是一个建立在全文搜索框架 Apache Lucene 基础上的搜索引擎。相较 Solr 等其他搜索引擎框架，Elastic Search 具有较好的实时分析能力，并提供分布式实时文件存储功能。在 Elastic Search 中，所有的对象都是以文档的形式进行组织，文档中的每一个字段都可以编入索引。同时，Elastic Search 还提供了友好的用户接口，支持 JSON 格式的序列化传输方式。作为分布式搜索引擎，Elastic Search 同样提供了集群化、分片、复制以及高可用性等特点。

Elastic Search 的简化版整体架构如图 7-15 所示，其中的 Gateway 组件代表 Elastic Search 索引的持久化存储方式。Elastic Search 边建索引边搜索速度没有太大变化，因为索引内容首先通过 Gateway 保存在内存中，内存不够时再持久化到硬盘，同时还有一个队列在空闲时自动把索引写到硬盘。关于 Elastic Search 的全面介绍不是本书的重点，读者可以参考相关资料<sup>[21]</sup>做进一步了解。

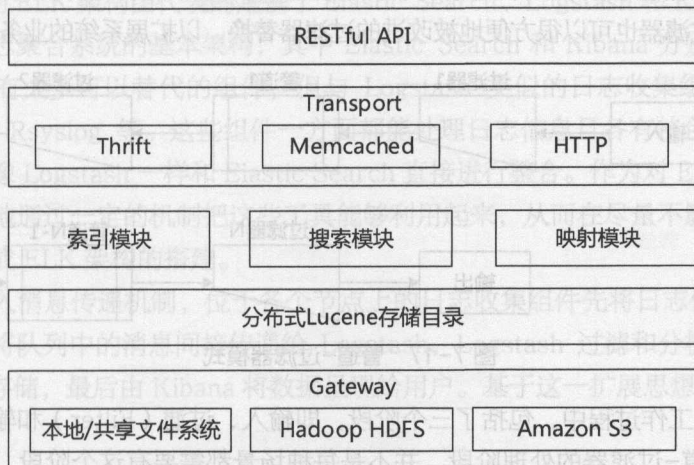


图 7-15 Elastic Search 简化版架构图

### (2) Logstash

Logstash 是一个具有实时渠道能力的数据收集引擎，可以对日志进行收集、过滤，并将其存储供搜索引擎等后续工具使用。Logstash 的主要特点在于几乎可以访问任何数据，可以和多种外部应用结合并支持弹性扩展。

Logstash 的基本结构见图 7-16，可以看到它由 Shipper、Broker 和 Indexer 三个组件所构成。其中 Shipper 用于发送日志数据；Broker 用于收集数据，缺省使用内置的 Redis；而 Indexer 则负责将数据写入 Elastic Search。



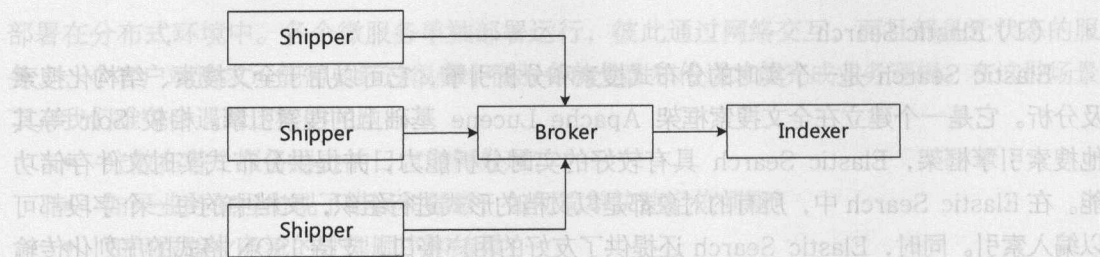


图 7-16 Logstash 基本组成

Logstash 使用管道-过滤器（Pipe-Filter）模式进行日志的搜集处理和输出，管道-过滤器模式结构示意图参考图 7-17。管道-过滤器结构将数据流处理分为几个顺序的步骤来进行，一个步骤的输出是下一个步骤的输入，每个步骤的处理由一个过滤器来实现。每个过滤器独立完成自己的任务，不同过滤器之间不需要交互。在管道-过滤器结构中，数据输出的最终结果与各个过滤器执行的顺序无关。这些特性允许将系统的输入和输出看作是各个过滤器行为的简单组合。独立的过滤器能够减小组件之间的耦合程度，我们可以很容易地将新过滤器添加到现有的系统之中，原有过滤器也可以很方便地被改进的过滤器替换，以扩展系统的业务处理能力。

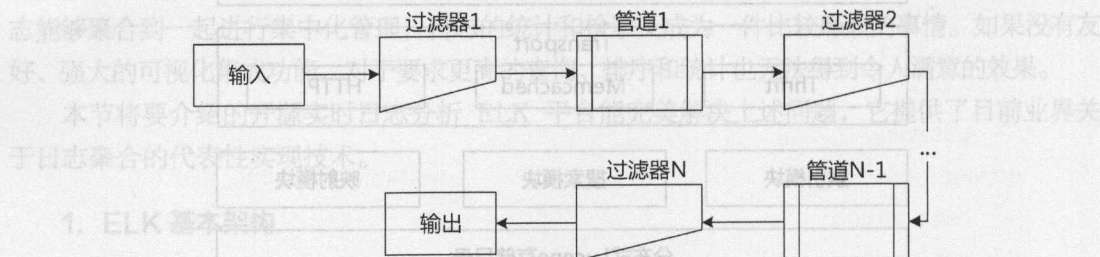


图 7-17 管道-过滤器模式

在 Logstash 工作过程中，包括了三个阶段，即输入、过滤（Filter）和输出，这里的过滤阶段就相当于管道-过滤器的处理阶段，并不是每种场景都需要有这个阶段，但可以根据需要设计并实现多个过滤器。Logstash 的功能结构参考图 7-18。

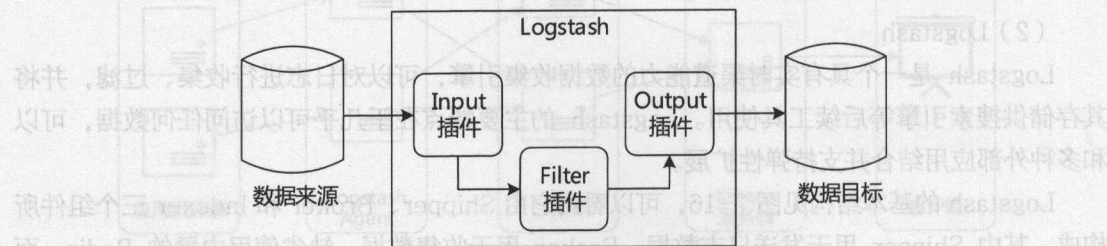


图 7-18 Logstash 功能结构图



Logstash 的强大之处在于每个阶段都有很多的插件配合工作，如 File、Elastic Search、Redis 等，而每个阶段也可以指定多种方式，例如，既可以输出到 Elastic Search 中，也可以指定到 stdout 在控制台打印。受益于管道-过滤器模式所具备的特点，Logstash 这种插件式的组织方式，使其具有高度的可扩展性和可定制性。更多关于 Logstash 的讨论可参考其官方网站<sup>[22]</sup>。

### (3) Kibana

Kibana 是一个开源的分析与可视化平台，和 Elastic Search 一起使用。我们可以使用 Kibana 搜索、查看、交互存放在 Elastic Search 索引里的数据。Kibana 提供各种不同的图表、表格、地图等表现形式，能够很轻松地展示高级数据分析与可视化效果。它利用 Elasticsearch 的 REST 接口来检索数据，不仅允许用户创建自己的数据定制仪表盘视图，还允许以特殊的方式查询和过滤数据。关于 Kibana 的最新信息可参考其官方网站 (<https://www.elastic.co/products/kibana>)。

## 2. ELK 架构扩展

图 7-14 中的 ELK 架构图代表的是基于 Elastic Search、Logstash 和 Kibana 这三个开源工具所构建的日志聚合系统的基本架构，其中 Elastic Search 和 Kibana 分别作为索引的存储和数据的展示没有太多可以替代的组件，但与 Logstash 类似的日志收集组件却有很多，如 Scribe、Flume、Rsyslog 等。这些组件一方面都能处理日志信息且各有特色，但另一方面，它们不一定能够像 Logstash 一样和 Elastic Search 直接进行整合。作为对 ELK 基本架构的扩展，我们也系统地通过一定的机制把这些工具能够利用起来，从而在尽量不影响现有日志处理方案的前提下完成 ELK 架构的搭建。

我们可以引入消息传递机制，位于各个节点上的日志收集组件先将日志传递给 Kafka 等消息中间件，并将队列中的消息间接传递给 Logstash，Logstash 过滤和分析后将数据传递给 Elastic Search 存储，最后由 Kibana 将数据呈现给用户。基于这一扩展思想的 ELK 架构图如图 7-19 所示。

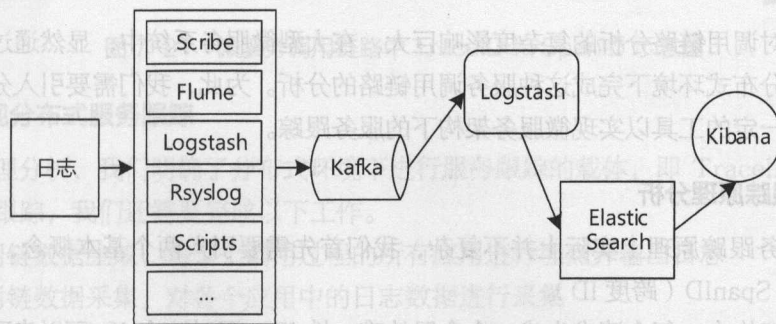


图 7-19 ELK 扩展架构图



这种扩展架构适合于较大集群的解决方案，通过引入消息传递机制，均衡了网络传输，通过消息中间件的存储转发机制降低网络阻塞尤其是丢失数据的可能性。同时也可以把各个日志收集工具统一集成到消息中间件从而实现系统的扩展性。

### 7.3.2 服务跟踪

微服务架构上基于业务划分服务并对外暴露服务访问接口。在中大型系统中，可能需要很多个服务协同才能完成一个接口功能，如果链路上任何一个服务出现问题或者网络超时，都会导致接口调用失败。随着业务的不断扩张，服务之间互相调用关系会越来越复杂。图 7-20 展示了从请求到响应的典型服务调用链路。我们看到当用户发起一个请求时，首先到达前端 A 服务，然后分别对 B 服务和 C 服务进行 RPC 调用；B 服务处理完给 A 服务做出响应，但是 C 服务还需要和后端的 D 服务、E 服务和 F 服务交互之后才能返还给 A 服务，最后由 A 服务来响应用户的请求。

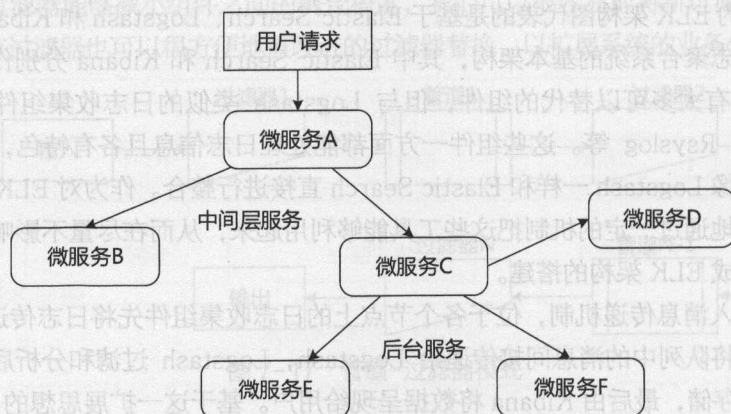


图 7-20 微服务调用链路示意图

服务数量对调用链路分析的复杂度影响巨大。在大型微服务系统中，显然通过人工手段已经完全不能在分布式环境下完成这种服务调用链路的分析。为此，我们需要引入分布式服务跟踪机制并借助一定的工具以实现微服务架构下的服务跟踪。

#### 1. 服务跟踪原理分析

分布式服务跟踪原理上实际上并不复杂，我们首先需要引入两个基本概念，即 TraceID（跟踪 ID）和 SpanID（跨度 ID）。

在微服务架构中，每个请求生成一个全局的唯一性 ID，通过这个 ID 可以串联起整个调用链，也就是说在分布式系统内部流转时，系统需要始终保持传递该唯一性 ID，直到请求返



回。这个唯一性 ID 就是 TraceID。

除了 TraceID 外,我们还需要 SpanID。当请求到达各个服务组件时,通过 SpanID 来标识它的开始、具体过程和结束。对于每个 Span 而言,它必须有开始和结束两个节点,通过记录开始 Span 和结束 Span 的时间戳统计该 Span 的时间延迟。

整个调用过程中每个请求都要透传 TraceID 和 SpanID。每个服务将该次请求附带的 TraceID 和 SpanID 作为 PSpanID (父 SpanID) 进行记录,并且生成自己的 SpanID。一个没有 PSpanID 的 Span 即为根 Span,可以看成调用链入口。所以要查看某次完整的调用只需根据 TraceID 查出所有调用记录,然后通过 PSpanID 和 SpanID 组织起整个调用父子关系。

关于 TraceID 和 SpanID 的关系我们可以通过图 7-21 做进一步解释。在图 7-21 中,我们通过四种关键事件记录服务的客户端请求和服务器响应过程,分别是 cs (Client Send, 客户端发送)、sr (Server Receive, 服务器接收)、ss (Server Send, 服务器发送) 和 cr (Client Receive, 客户端接收),它们构成了客户端和服务器对一次请求处理的闭环。

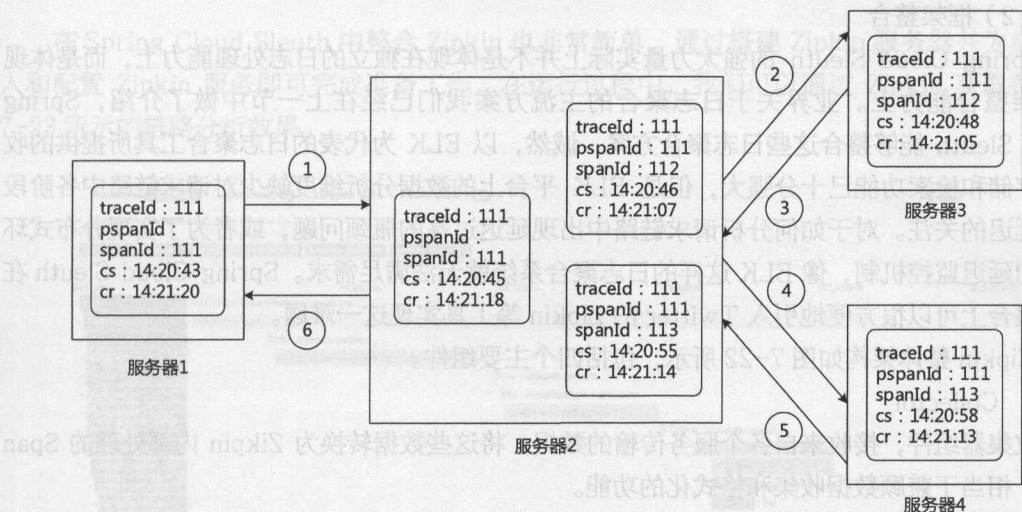


图 7-21 微服务调用链路中 TraceID 和 SpanID 示意图

## 2. 实现分布式服务跟踪

通过原理分析,我们明确了分布式环境下进行服务跟踪的载体,即 TraceID 和 SpanID。但实现服务跟踪,我们还需要完成以下工作。

- 调用链数据生成,对整个调用过程的所有应用进行埋点并输出日志
- 调用链数据采集,对各个应用中的日志数据进行采集
- 调用链数据存储及查询,对采集到的数据进行存储,由于日志数据量一般都很大,不仅要能对其存储,还需要能提供快速查询



- 对采集到的日志数据进行各种指标运算, 并将运算结果保存起来
- 告警功能, 提供各种阈值警告功能

这些工作不是简单的一个工具和框架能全部完成的, 我们也不想自己从无到有实现这样一整套解决方案。幸好在 Spring Cloud 中存在一个组件能够帮助我们简化实现, 这个组件就是 Spring Cloud Sleuth。

### (1) Spring Cloud Sleuth

通过将 Spring Cloud Sleuth 添加到系统的类路径, 系统便会自动建立日志收集渠道, 包括基于 Apache Kafka、RabbitMQ 或其他 Spring Cloud Stream 捆绑程序等消息传递技术发送的请求; Spring MVC 控制器接收的 HTTP 请求; 通过 Netflix Zuul 微代理发送的请求; 使用 RestTemplate 发出的请求等。同时 Spring Cloud Sleuth 可以设置实用的日志格式来输出 TraceID 和 SpanID。我们可以利用诸如 Logstash 等日志发布组件将日志发布到 Elastic Search 等日志分析工具中进行处理。

### (2) 框架整合

Spring Cloud Sleuth 的强大力量实际上并不是体现在独立的日志处理能力上, 而是体现在框架整合能力上。业界关于日志聚合的主流方案我们已经在上一节中做了介绍, Spring Cloud Sleuth 能够整合这些日志聚合方案。诚然, 以 ELK 为代表的日志聚合工具所提供的收集、存储和检索功能已十分强大, 但是 ELK 平台上的数据分析维度缺少对请求链路中各阶段时间延迟的关注。对于如何分析请求链路中出现延迟过高的瓶颈问题, 或者为了实现分布式环境中的延迟监控机制, 像 ELK 这样的日志聚合系统就无法满足需求。Spring Cloud Sleuth 在框架整合上可以很方便地引入 Twitter 的 Zipkin 等工具实现这一难题。

Zipkin 整体架构如图 7-22 所示, 包括四个主要组件。

- Collector

收集器组件, 接收来自各个服务传输的数据, 将这些数据转换为 Zipkin 内部处理的 Span 格式, 相当于兼顾数据收集和格式化的功能。

- Storage

存储组件, 存储收集过来的数据, 当前支持 Cassandra、Redis、HBase、MySQL、PostgreSQL、SQLite 等, 默认存储在内存中。

- RESTful API

API 组件, 负责查询 Storage 中存储的数据, 提供简单的 RESTful API 获取数据, 主要提供给 Web UI 使用。

- Web UI

UI 组件, 提供简单的 Web 界面, 基于 API 组件的上层应用, 可以方便而直观地查询和分析跟踪信息。



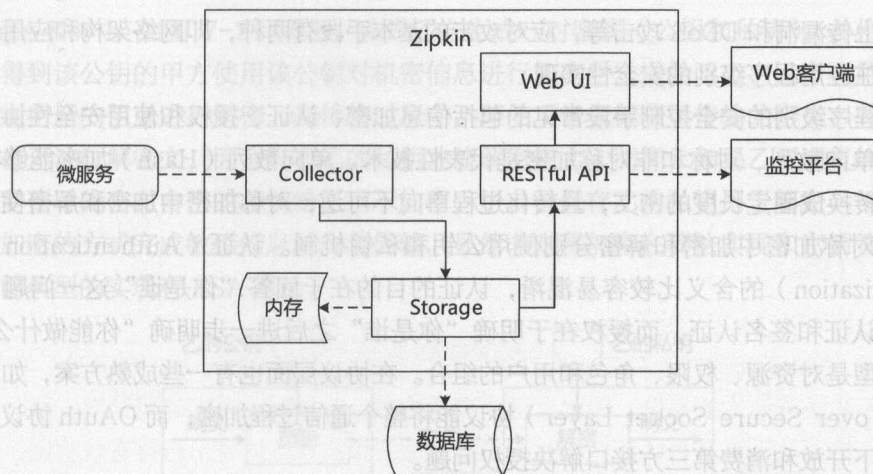


图 7-22 Zipkin 基本架构

在 Spring Cloud Sleuth 中整合 Zipkin 也非常简单，通过搭建 Zipkin 服务器并为应用引入和配置 Zipkin 服务即可完成准备工作。在运行过程中，我们可以通过 Zipkin 获取类似图 7-23 所示的链路分析效果。

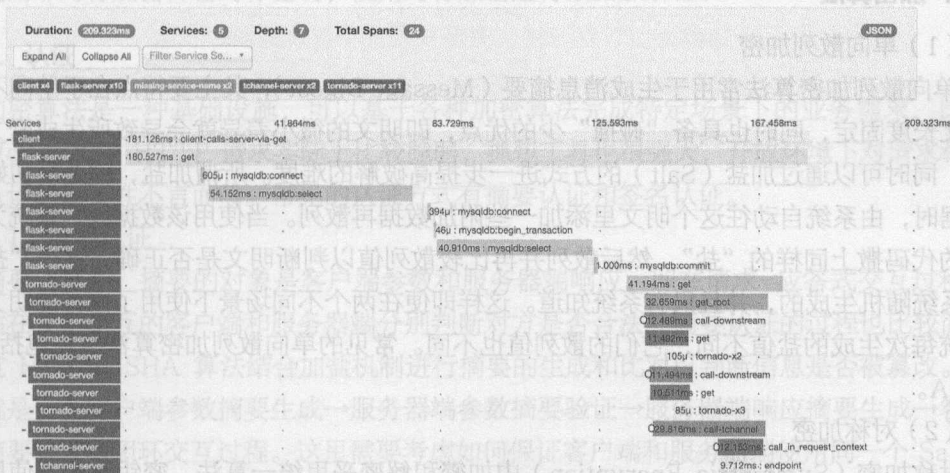


图 7-23 Zipkin 服务调用链路分析示例图（来自 Zipkin 官网）

## 7.4 服务安全

我们的系统可能面临着各种潜在的网络攻击手段，包括 XSS 攻击、注入攻击、CSRF 攻



击、文件上传漏洞和 DDoS 攻击等，应对攻击的基本手段有两种，即网络架构和应用程序。本书主要关注应用程序级别的安全性实现。

应用程序级别的安全控制手段常见的包括信息加密、认证、授权和使用安全性协议。信息加密上，单向散列、对称和非对称加密是代表性技术。单向散列（Hash）加密能够将不同长度的明文转换成固定长度的密文，且转化过程单向不可逆；对称加密中加密和解密使用同一密钥，而非对称加密中加密和解密分别使用公钥和私钥机制。认证（Authentication）和授权（Authorization）的含义比较容易混淆，认证的目的在于回答“你是谁”这一问题，常见的包括摘要认证和签名认证，而授权在于明确“你是谁”之后进一步明确“你能做什么”，通用的授权模型是对资源、权限、角色和用户的组合。在协议层面也有一些成熟方案，如 HTTPS（HTTP over Secure Socket Layer）协议能将整个通信过程加密，而 OAuth 协议则通过分布式环境下开放和消费第三方接口解决授权问题。

#### 7.4.1 通用安全性技术

通用的安全性实现技术包括加密、认证和协议。

##### 1. 加密算法

###### （1）单向散列加密

单向散列加密算法常用于生成消息摘要（Message Digest），其主要特点在于单向不可逆和密文长度固定，同时也具备“碰撞”少的优点，即明文的微小差异就会导致所生成密文完全不同。同时可以通过加盐（Salt）的方式进一步提高破解的难度。所谓加盐，就是在初始化明文数据时，由系统自动往这个明文里添加一些附加数据再散列。当使用该数据时，系统为用户提供的代码撒上同样的“盐”，然后散列并再比较散列值以判断明文是否正确。这个“盐”值是由系统随机生成的，并且只有系统知道。这样即便在两个不同场景下使用了同样的明文，由于系统每次生成的盐值不同，它们的散列值也不同。常见的单向散列加密算法实现包括 MD5 和 SHA。

###### （2）对称加密

对称加密（Symmetric Encryption）中加密和解密采用统一算法，密钥对称。使用对称密钥的优点在于简单、高效、长密钥难破解，但需要确保密钥交换过程的安全性。DES 和 AES 算法是目前对称加密的主要实现方式。

###### （3）非对称加密

区别于对称加密在加密和解密时使用同一密钥，非对称加密（Asymmetric Encryption）需要两个密钥来进行加密和解密，这两个密钥分别称为公钥（Public Key）和私

仅供对书籍质量进行鉴定甄别！是否为购买正版实体书提供依据！！  
非卖品！！严禁（售卖和上传互联网平台）！！



钥 (Private Key)。如图 7-24 所示, 首先乙方生成一对密钥 (公钥和私钥), 并将公钥向甲方公开, 得到该公钥的甲方使用该公钥对机密信息进行加密后发送给乙方, 乙方再用自己保存的私钥对加密后的信息进行解密。在传输过程中, 即使攻击者截获了传输的密文, 并得到了乙的公钥, 也无法破解密文, 因为只有乙的私钥才能解密密文。同样, 如果乙要回复加密信息给甲, 那么需要甲先公布自己的公钥给乙用于加密, 甲自己保存乙的私钥用于解密, 甲乙之间使用非对称加密的方式完成敏感信息的安全传输。显然非对称加密在算法实现和密钥管理上比较复杂, 目前典型的实现有 RSA 算法。

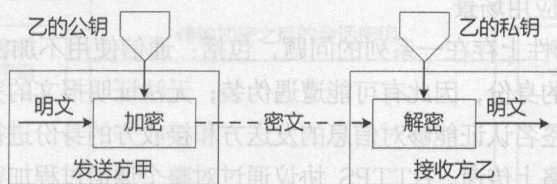


图 7-24 非对称加密过程示意图

以上三种加密算法各有其应用场景, 单向散列加密主要用于生成信息摘要和随机数, 对称加密用于通信加密, 而非对称加密用于信息安全传输。同时, 算法之间也可以混合使用, 例如可以使用非对称加密传输对称密钥, 使用对称加密进行数据加解密。

## 2. 认证

加密算法是一种基础设施, 可以基于各种加密算法完成特定场景下的业务需求, 比如认证。Web 应用中 HTTP 请求实际上比较脆弱, 抓包工具也很强大, 开放环境下对请求和响应进行认证的重要性不言而喻。常见的认证方式有摘要认证和签名认证。

### (1) 摘要认证

摘要认证中, 摘要的对象是客户端参数和服务器端响应, 即在请求-应答式交互过程中, 需要站在发生交互的客户端和服务器端分别判断对方是否合法。摘要认证的过程也比较简单, 即通过 MD5 或 SHA 算法结合加盐机制进行摘要的生成和比对以判断信息是否被篡改。整个认证就是一个客户端参数摘要生成→服务器端参数摘要验证→服务器端响应摘要生成→客户端响应摘要验证的闭环交互过程。这里需要考虑如何保证客户端和服务器端使用同一个 Salt, 服务器端可以在认证之前通过某种方式把 Salt 发给需要接入的客户端。

### (2) 签名认证

摘要认证的主要问题就是如何防止 Salt 泄露, 而签名认证则不使用 Salt, 基本思路是使用非对称加密算法加密数字摘要, 是混合算法的一种具体应用。同摘要认证一样, 签名认证也通过客户端参数签名生成→服务端参数签名验证→服务端响应签名生成→客户端响应签名验证完成闭环。摘要认证中通常使用 MD5withRSA 和 SHA1withRSA 等算法组合。



## 7.4.2 安全性协议

本节我们将介绍在安全性协议领域非常典型的两种技术体系，一个是 HTTPS 协议，另一个是 OAuth 协议。这两个协议有着完全不同的应用场景，但都是微服务架构中实现安全性协议的基本手段。

### 1. HTTPS

#### (1) HTTPS 协议应用场景

HTTP 协议在安全性上存在一系列的问题，包括：通信使用不加密的明文，内容可能会被窃听；不验证通信方的身份，因此有可能遭遇伪装；无法证明报文的完整性，所以有可能已遭篡改等。摘要认证和签名认证能够对信息的发送方和接收方的身份进行有效验证，但敏感信息仍然以明文方式在网络上传递。HTTPS 协议通过对整个通信过程加密的方式确保敏感信息不被泄漏，可以把 HTTPS (HTTP over SSL) 看作是 HTTP 协议加上 SSL/TLS 的结合体 (见图 7-25)。

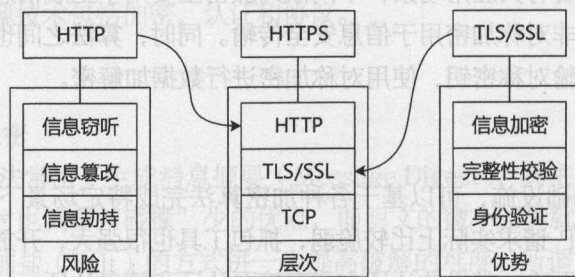


图 7-25 HTTPS=HTTP+SSL/TLS

#### (2) HTTPS 协议工作流程

SSL/TLS 协议的基本思路是采用公钥加密法，也就是说，客户端先向服务器端索要公钥，然后用公钥加密信息，服务器收到密文后，用自己的私钥解密。因为需要对整个通道的所有信息进行加密，而非对称加密计算量太大导致加密效率过低，SSL/TLS 协议的做法是针对每一次会话 (Session)，客户端和服务端都生成一个会话密钥 (Session Key)，用它来加密信息。由于会话密钥是对称加密，所以运算速度非常快，而服务器公钥只用于加密会话密钥本身，这样就减少了加密运算的消耗时间。从加密的角度上讲，以上的设计思路体现的是采用对称加密和非对称加密两者并用的混合加密机制。这样 SSL/TLS 协议的基本过程就分成三步，即客户端向服务器端索要并验证公钥→双方协商生成对话密钥→双方采用对话密钥进行加密通信，HTTPS 的整体工作流程参考图 7-26，因为不同的节点之间所产生的会话密钥不同，从而可以保证信息只能通信双方获取。



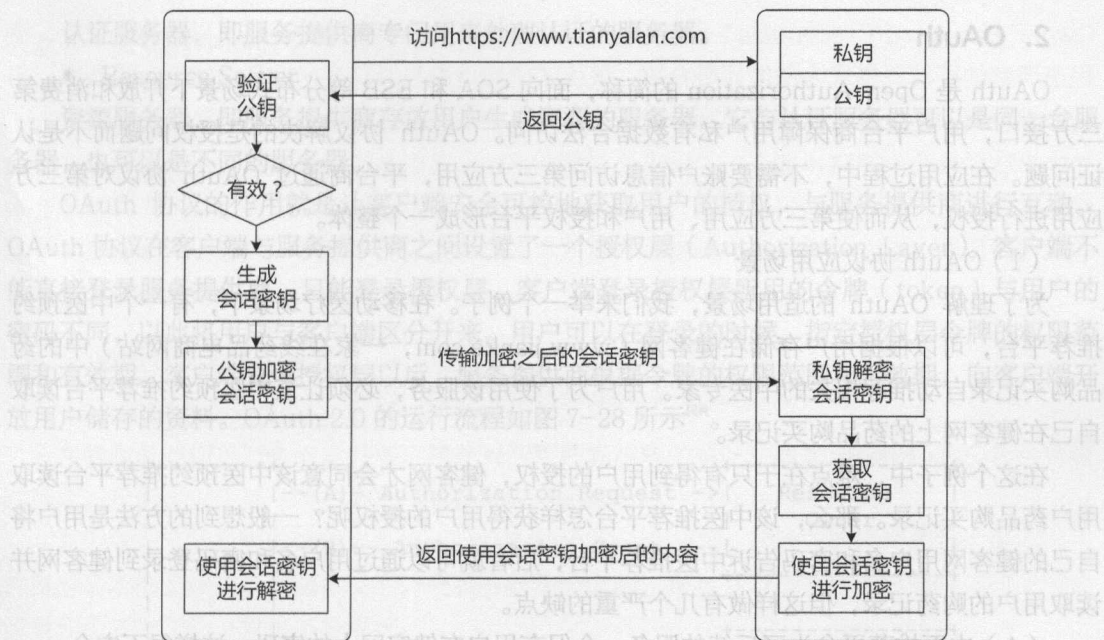


图 7-26 HTTPS 协议工作流程图

图 7-26 中, 客户端向服务器端索要并验证公钥和双方协商生成对话密钥这两步又被称为握手阶段 (Handshake), 见图 7-27。SSL/TLS 协议的握手阶段非常复杂, JSSE (Java Secure Socket Extension, Java 安全套接字扩展) 是 SSL 和 TLS 的纯 Java 实现, 通过它可以透明地提供数据加密、服务器认证、信息完整性等功能, 如同使用普通的套接字一样使用安全套接字。通过 JSSE, 开发者可以很轻松地将 SSL 协议整合到应用程序中, 并且 JSSE 能将安全隐患降到最低点。

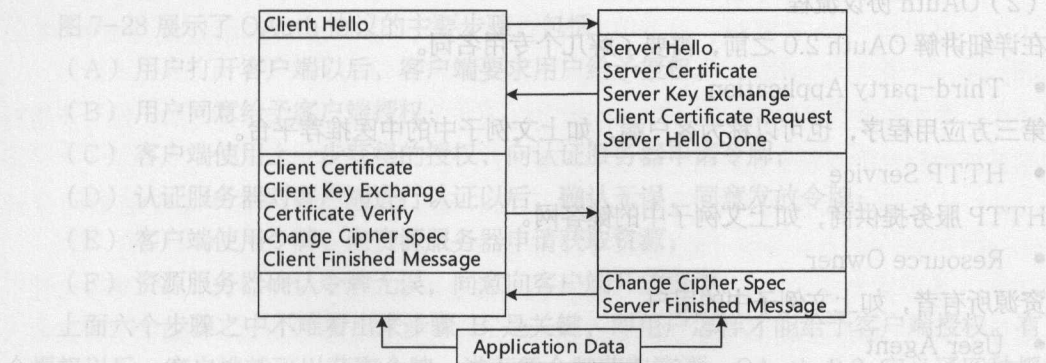


图 7-27 SSL/TLS 协议握手阶段



## 2. OAuth

OAuth 是 Open Authorization 的简称，面向 SOA 和 ESB 等分布式场景下开放和消费第三方接口，用户平台商保障用户私有数据合法访问。OAuth 协议解决的是授权问题而不是认证问题。在应用过程中，不需要账户信息访问第三方应用，平台商通过 OAuth 协议对第三方应用进行授权，从而使第三方应用、用户和授权平台形成一个整体。

### (1) OAuth 协议应用场景

为了理解 OAuth 的适用场景，我们来举一个例子。在移动医疗场景下，有一个中医预约推荐平台，可以根据用户存储在健客网（www.jianke.com，一家在线药品电商网站）中的药品购买记录自动推荐相关的中医专家。用户为了使用该服务，必须让该中医预约推荐平台读取自己在健客网上的药品购买记录。

在这个例子中，难点在于只有得到用户的授权，健客网才会同意该中医预约推荐平台读取用户药品购买记录。那么，该中医推荐平台怎样获得用户的授权呢？一般想到的方法是用户将自己的健客网用户名和密码告诉中医推荐平台，后者就可以通过用户名和密码登录到健客网并读取用户的购药记录，但这样做有几个严重的缺点。

(A) 中医推荐平台为了后续的服务，会保存用户在健客网上的密码，这样很不安全；

(B) 中医推荐平台拥有了获取用户储存在健客网上所有资料的权力，用户没法限制中医推荐平台获得授权的范围和有效期；

(C) 用户只有修改密码，才能收回赋予中医推荐平台的权力，但是这样做会使得其他所有获得用户授权的类似中医推荐平台的第三方应用程序全部失效；

(D) 只要有一个第三方应用程序被破解，就会导致用户密码泄漏，以及所有被密码保护的数据产生泄漏。

OAuth 协议的诞生就是为了解决以上问题。

### (2) OAuth 协议流程

在详细讲解 OAuth 2.0 之前，需要了解几个专用名词。

#### • Third-party Application

第三方应用程序，也可以称为客户端，如上文例子中的中医推荐平台。

#### • HTTP Service

HTTP 服务提供商，如上文例子中的健客网。

#### • Resource Owner

资源所有者，如上文例子中的用户。

#### • User Agent

用户代理，很多场合指的是浏览器。

#### • Authorization Server



认证服务器，即服务提供商专门用来处理认证的服务器。

- Resource Server

资源服务器，即服务提供商存放用户生成资源的服务器。它与认证服务器可以是同一台服务器，也可以是不同的服务器。

OAuth 协议的作用就是让客户端安全可控地获取用户的授权，与服务提供商进行互动。OAuth 协议在客户端与服务提供商之间设置了一个授权层（Authorization Layer）。客户端不能直接登录服务提供商，只能登录授权层，客户端登录授权层所用的令牌（token）与用户的密码不同，以此将用户与客户端区分开来。用户可以在登录的时候，指定授权层令牌的权限范围和有效期。客户端登录授权层以后，服务提供商根据令牌的权限范围和有效期，向客户端开放用户储存的资料。OAuth 2.0 的运行流程如图 7-28 所示<sup>[23]</sup>。

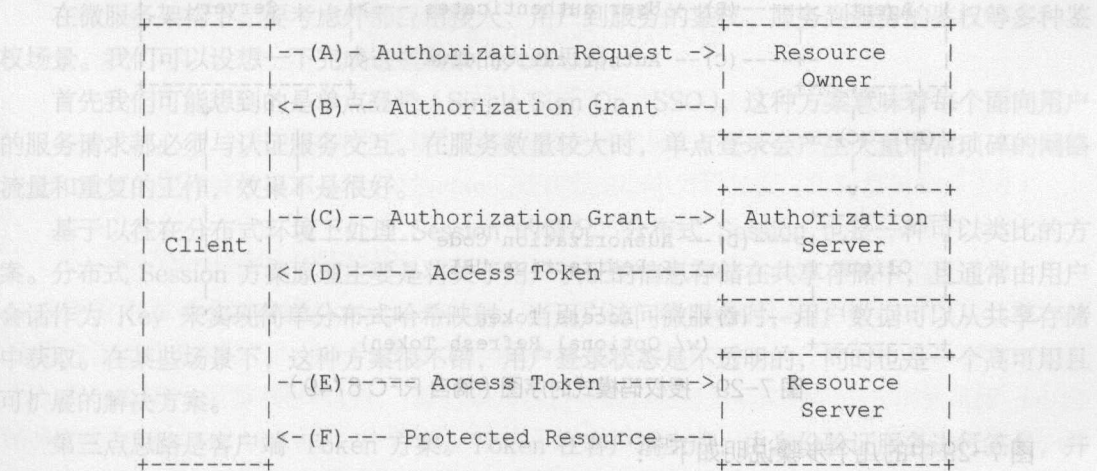


图 7-28 OAuth 协议时序图（摘自 RFC 6749）

图 7-28 展示了 OAuth 协议的主要步骤，包括：

- (A) 用户打开客户端以后，客户端要求用户给予授权；
- (B) 用户同意给予客户端授权；
- (C) 客户端使用上一步获得的授权，向认证服务器申请令牌；
- (D) 认证服务器对客户端进行认证以后，确认无误，同意发放令牌；
- (E) 客户端使用令牌，向资源服务器申请获取资源；
- (F) 资源服务器确认令牌无误，同意向客户端开放资源。

上面六个步骤之中不难看出步骤 B 是关键，即用户怎样才能给予客户端授权。有了这个授权以后，客户端就可以获取令牌，进而凭令牌获取资源。OAuth 2.0 定义了四种授权方式，即授权码模式（Authorization Code）、简化模式（Implicit）、密码模式（Resource



Owner Password Credentials) 和客户端模式 (Client Credentials)。这里以功能最完整、流程最严密的授权码模式为例做简要介绍，该模式的特点就是通过客户端系统的后台服务器，与服务提供商的认证服务器进行交互。图 7-29 展示了授权码模式的典型时序图。

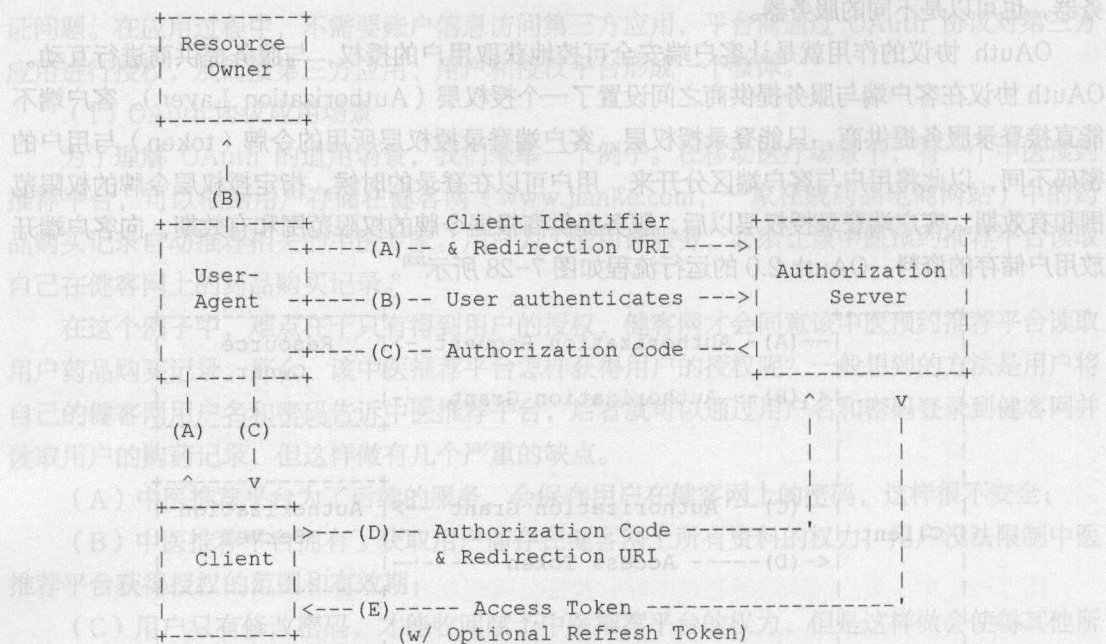


图 7-29 授权码模式时序图 (摘自 RFC 6749)

图 7-29 中的几个步骤说明如下<sup>[23]</sup>：

- (A) 用户访问客户端，后者将前者导向认证服务器；
- (B) 用户选择是否给予客户端授权；
- (C) 假设用户给予授权，认证服务器将用户导向客户端事先指定的重定向 URI (Redirection URI)，同时附上一个授权码；
- (D) 客户端收到授权码，附上早先的重定向 URI，向认证服务器申请令牌。这一步是在客户端系统的后台服务器上完成的，对用户不可见；
- (E) 认证服务器核对授权码和重定向 URI，确认无误后，向客户端发送访问令牌 (Access Token) 和更新令牌 (Refresh Token)。

OAuth 协议在实现上体系比较复杂，综合应用摘要认证、签名认证、HTTPS 等安全性手段，需要提供 Token 生成和校验、分布式 Session 和公私钥管理等功能，同时需要开发者入驻并进行权限粒度控制。一般我们可以借助 Apache Oltu、Spring Security for OAuth 等工具来搭建 OAuth 平台。



### 7.4.3 微服务中的安全性设计

本节前面所介绍的通用安全性技术和协议在微服务架构中同样适用，但在微服务环境下，实现安全性也要面临特定场景下的挑战。因为在微服务架构中，各服务部署在分布式环境中的多套容器之内。各服务接口不再存在于本地，而是通过 HTTP 进行远程接入。这里的挑战就在于我们要如何验证用户并在不同微服务之间以对称方式完成登录信息的传递，同时还要想办法让各个微服务完成对用户的授权。

#### 1. 微服务安全性实现方案

##### (1) 微服务安全策略

在微服务架构下，要考虑外部应用接入、用户到服务的鉴权、服务到服务的鉴权等多种鉴权场景。我们可以设想一下完成这些场景的大致思路。

首先我们可能想到的是单点登录（Single Sign On, SSO）。这种方案意味着每个面向用户的服务请求都必须与认证服务交互。在服务数量较大时，单点登录会产生大量非常琐碎的网络流量和重复的工作，效果不是很好。

基于以往在分布式环境下处理 Session 的情况，分布式 Session 也是一种可以类比的方案。分布式 Session 方案原理主要是将关于用户认证的信息存储在共享存储中，且通常由用户会话作为 Key 来实现简单分布式哈希映射。当用户访问微服务时，用户数据可以从共享存储中获取。在某些场景下，这种方案很不错，用户登录状态是不透明的，同时也是一个高可用且可扩展的解决方案。

第三点思路是客户端 Token 方案。Token 在客户端生成，由身份验证服务进行签名，并且必须包含足够的信息，以便可以在所有微服务中建立用户身份。Token 会附加到每个请求上，为微服务提供用户身份验证。对于客户端 Token 的编码方案，我们可以使用 JSON Web Tokens (JWT)，它足够简单且开发库支持程度也比较好。

我们在本书 4.5 节中介绍 API 网关时提到网关的一个作用就是进行安全控制，所以我们可以把客户端 Token 与 API 网关结合起来。这个方案意味着所有请求都通过网关，从而有效地隐藏了微服务。本节后续内容将对基于 Token 的认证方案做进一步介绍。

##### (2) 基于 Token 的认证方案

随着 RESTful API 和微服务的兴起，基于 Token 的认证现在已经越来越普遍。Token 和 Session ID 不同，并非只是一个 Key。Token 一般会包含用户的身份信息，通过验证 Token 就可以完成身份校验。图 7-30 展示了基于 Token 的基本工作机制。



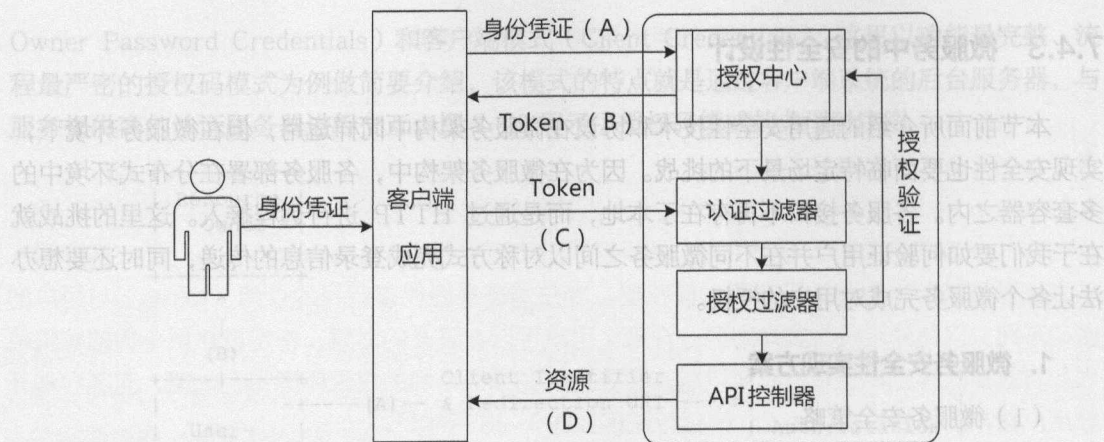


图 7-30 Token 机制基本示意图

如图 7-30 所示的基于 Token 的验证流程如下所示。

- (A) 用户使用包含用户名和密码的身份凭证从客户端发起资源请求；
- (B) 后端接受请求，通过授权中心生成有效 Token 字符串，返回给客户端；
- (C) 客户端获得 Token 后，再次发出资源请求；
- (D) 后端接受带 Token 的请求，通过授权中心获取相关资源并返回给客户端。

基于 Token 认证的好处包括：服务端无状态，Token 机制在服务端不需要存储 Session 信息，因为 Token 自身包含了用户的所有相关信息；性能较好，因为在验证 Token 时不用再去访问数据库或者远程服务进行权限校验；支持跨程序调用，Cookie 是不允许跨域访问的，Token 则不存在这个问题。实际上，上一节中介绍的 OAuth 协议也是一种基于 Token 的实现方案，我们不再具体展开。本节将重点介绍的是 JWT 方案。

JSON Web Token (JWT) 是为了在网络应用环境间传递声明而制定的一种基于 JSON 的开放标准。根据 RFC 7519 的描述<sup>[24]</sup>，JSON Web Token 以一种紧凑的、URL 安全的方式在双方之间传输安全性声明。JWT 一般用来在身份提供者和服务提供者间传递被认证的用户身份信息，以便从资源服务器获取资源。同时，也可以增加一些额外信息用于处理复杂的业务逻辑。JSON Web Token 可以直接被用于认证，也可以被加密。JWT 认证基本流程如下所示。

- (A) 客户端调用登录接口，传入用户名和密码。
- (B) 服务端请求身份认证中心，确认用户名密码正确。
- (C) 服务端创建 JWT，返回给客户端。
- (D) 客户端拿到 JWT，进行存储。可以存储在缓存中，也可以存储在数据库中。如果是浏览器，还可以存储在 Cookie 中。在后续请求中，在 HTTP 请求头中加上 JWT。
- (E) 服务端校验 JWT，校验通过后，返回相关资源和数据。



JWT 本身是由三段信息构成的, 第一段为头部 (Header), 第二段为载荷 (Payload), 第三段为签名 (Signature)。每一段内容都是一个 JSON 对象, 将每一段 JSON 对象采用 BASE64 编码, 再将编码后的内容用 “.” 链接一起就构成了 JWT 字符串: header.payload.signature。

JWT 的优点在于支持跨语言, JSON 格式提供了语言无关性; 同时, Token 占用字节小, 便于传输。关于 JWT 需要关注一个问题, 就是 Token 的注销方式。由于 Token 不存储在服务端, 当用户注销时, Token 的有效时间可能还没有到, 所以如何在用户注销的同时让 Token 失效是实现上的难点。一般有如下几种方式。

- Token 存储在 Cookie 中  
这样客户端注销时, 自然可以清空掉 Token。
- Token 存放到分布式缓存  
注销时, 将 Token 注销状态存放到诸如 Redis 这样的分布式缓存中。每次校验 Token 时先从缓存中检查该 Token 是否已注销。
- 采用短期令牌

通过 Token 过期机制一定程度上能降低注销后 Token 可用性的风险。例如, 将令牌有效期设置为 20 分钟, 过了 20 分钟该 Token 就自动失效。

## 2. Spring Cloud Security

作为 Spring Cloud 中的一员, Spring Cloud Security 是对微服务架构中所面临的安全性问题进行抽象并实现的工具。Spring Cloud Security 具备以下特点: 基于 OAuth2 和 OpenID 协议的可配置的 SSO 登录机制, 基于 Token 保障资源访问安全, 另外, 引入 UAA (User Account and Authentication, 用户账户和认证) 服务, UAA 是一个 Web 服务, 用于管理账户、OAuth2 客户端和用户用于鉴权的 Issue Token (问题令牌), UAA 实现了 OAuth2 授权框架和基于 JWT 的 Token 机制。

UAA 主要功能是封装 OAuth2 协议, 当用户访问客户端应用时, 生成并发放 Token 给目标客户端。在认证对象方面, UAA 支持用户、客户端以及目标资源服务器; 认证类型主要有授权码模式、密码模式以及客户端模式; 而认证范围则是通过一个命名的参数附加到 Token 上的方式进行实现。

## 7.5 本章小结

在本章中, 我们从服务测试、服务交付和部署、服务监控以及服务安全等四个方面对如何建设微服务架构的管理体系展开讨论。

针对服务测试, 微服务架构在实现上一方面关注于单元测试和集成测试等测试方法, 另一



方面,由于微服务之间存在的相互依赖关系,我们倾向于使用消费者驱动的契约测试方法来对这种依赖关系进行统一管理,本章对这一特定的测试方面做了介绍,并将 Spring Cloud Contract 作为其实现方案。

以 Docker 为代表的容器技术的兴起为微服务架构的落地提供了基础设施上的保障,本章同样对 Docker 以及用于微服务编排的 Docker Compose 做了介绍,并结合配置管理和持续集成讨论了微服务交付的方法和实践。

服务监控的一个重点工作是日志聚合,通过目前主流的 ELK 架构能够有效实现对日志的管理并从日志中找到分析问题和解决问题的方法。同时,我们也需要对当前系统中的服务运行状态进行跟踪和监控,在 Spring Cloud 中同样提供了专门用于分布式服务跟踪的 Spring Cloud Sleuth 组件。

最后,本章还针对服务安全性设计做了分析,在介绍常见的加密、授权、认证算法和安全性协议的基础上,给出了微服务架构实现安全性方面的特定技术和工具。

如图 7-80 所示的基于 Token 的验证流程如下所示。

(B) 后端接受请求,通过授权中心获取 Token 并返回给客户端。  
(C) 客户端获得 Token 后,再次发出资源请求。

(D) 后端接受带 Token 的请求,通过授权中心获取 Token 并返回给客户端。  
JSON Web Token (JWT) 是为了在网络应用环境间传递声明而设计的一种开放标准 (RFC 7519),它通常被用于身份验证和信息交换。JWT 是一种紧凑、自包含的字符串,它由三部分组成:头部 (Header)、载荷 (Payload) 和签名 (Signature)。

JSON Web Token (JWT) 是为了在网络应用环境间传递声明而设计的一种开放标准 (RFC 7519),它通常被用于身份验证和信息交换。JWT 是一种紧凑、自包含的字符串,它由三部分组成:头部 (Header)、载荷 (Payload) 和签名 (Signature)。

(A) 客户端调用登录接口,传入用户名和密码。  
(B) 服务端请求身份认证中心,验证用户名和密码是否正确。  
(C) 服务端生成 JWT,返回给客户端。

同时,JWT 还可以用于其他场景,如:跨域资源共享 (CORS)、API 网关鉴权等。在 JWT 的头部,通常包含元数据,如:令牌类型 (JWT)、版本号 (1.0) 等。在 JWT 的载荷,通常包含用户身份信息,如:用户名、角色等。在 JWT 的签名,通常使用 HMAC 算法进行签名,以确保数据的完整性和真实性。



## 第四篇 服务转型

### ◆ 本篇内容

本篇共有一章，从实际应用角度出发探讨如何在现有系统的基础上向微服务架构转型。

微服务架构转型需要过程和方法，一方面，可以应用常见的调整架构的技术，并分析微服务架构与现有系统的并存方法；另一方面，需要梳理微服务实施的最佳实践。微服务架构转型还需要考虑研发过程因素，为了实现微服务架构，产品管理、组织管理和研发文化都需要进行相应的转变。

作为本书的最后一部分内容，我们将通过一个具体案例来介绍微服务架构转型实施过程中的四个不同阶段以及各个阶段的实施要点。在第一阶段，重点关注业务功能梳理、业务边界划分、业务数据规划以及服务拆分和集成策略。在第二阶段，我们则需要梳理各个核心微服务，完成整体架构设计和技术体系的建立。在第三阶段，我们关注于过程，其中涉及产品管理、组织架构、研发文化的演变。而在最后的第四阶段，需要实现持续交付以及过程资产建设。

#### • 现有系统优化

侧重于对现有系统在架构上开展优化工作，包含综合整理共享库、共享服务、代码转移、代码冗余、提取新服务和重写服务等手段。



## 第 8 章

# 向微服务架构转型

服务监控的一个重点工作是日志聚合，通过目前主流的 ELK 架构能够有效实现对日志的管理并从日志中找到分析问题和解决问题的方法。同时，我们也

当下互联网应用发展日新月异，无论从产品形态还是技术实现上，我们都需要更快、更好地提供服务。在本书第 1 章中，我们提到了微服务架构在技术、业务和组织上的优势。对于很多现有系统而言，向微服务化转型也是一种顺应时代发展的客观需求。

本书第 3 章开篇提到架构轮回，任何一个系统的架构和实现演变到一定阶段时，势必会产生架构腐化，也势必需要进行架构的重构甚至推倒重来。一个架构设计的好坏与否往往影响着架构腐化的过程和时机。微服务架构被认为是一种可以推迟架构腐化的到来、降低系统重构成本的有效手段。

当一个系统出现以下症状时，我们可以认为已经到了向微服务架构转型的时候。

- 没有服务化

各产品系统独立开发，代码复用率低，系统之间互相调用且耦合严重，系统解耦及独立部署困难。

- 数据问题

应用间数据复制现象严重，因为数据不一致性而导致的系统问题时常发生。

- 基础组件

基础组件薄弱，日志、监控等基础设施类系统不完善。

- 功能模块定义

没有产品化思维，业务结构混乱，功能实现上包含大量重复性代码。

- 可靠性

在大容量访问下无法提供可靠性服务，间断性出现服务不可用现象，且很难找到针对性的解决方案。

本章结合全书前面各章中的内容，通过一个案例阐述向微服务架构转型中涉及的过程和方法，以及需要实现的产品管理、组织架构和研发文化上的转变。



## 8.1 微服务架构转型过程与方法

现有系统向微服务架构转型在实施上的一种方式就是推倒重来，这种方式相当于在没有任何历史包袱的条件下重新开发微服务架构体系，相对比较简单，这里不单独展开讨论。

另一种更常见，也更现实的方式就是采用逐步替换的策略实现微服务架构转型，本书3.1.2节中介绍的绞杀者模式就是这一策略的具体实现模式。在本章中，我们将采用如图8-1所示的微服务架构改造过程与方法来实现向微服务架构转型。

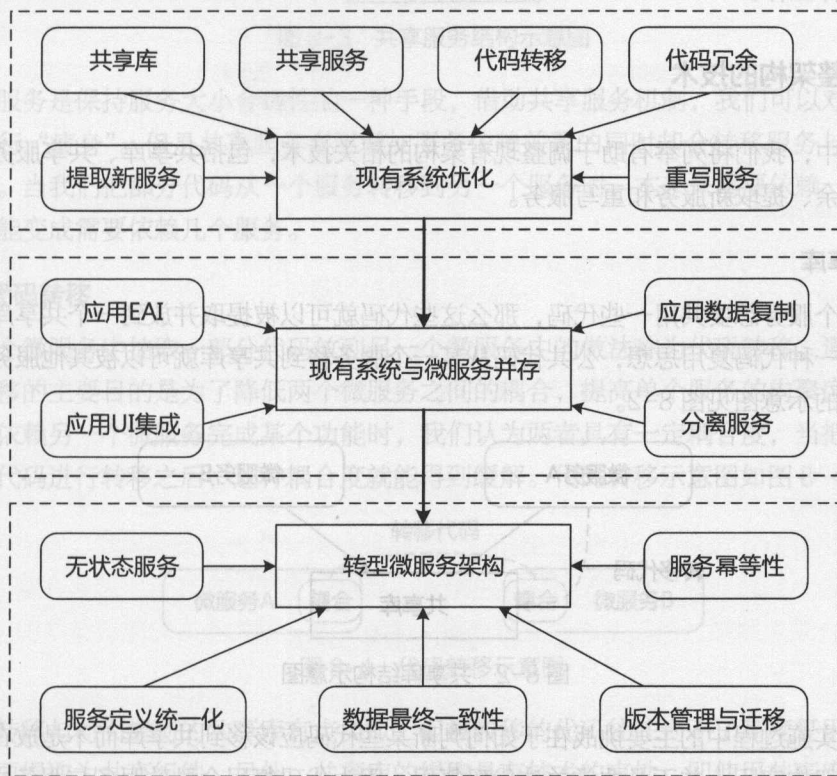


图 8-1 微服务架构改造过程与方法

在图8-1中，我们把转型过程分成三个阶段，每个阶段都会采用一些方式方法来推动这一转型过程。

- 现有系统优化

侧重于对现有系统在架构上开展优化工作，包含综合使用共享库、共享服务、代码转移、代码冗余、提取新服务和重写服务等手段。



## 第8章

- 现有系统与微服务并存

关注在转型过程中现有系统和微服务架构并存的这一过渡阶段，从应用 EAI（Enterprise Application Integration，企业应用集成）、应用数据复制、应用 UI 集成和分离服务等角度出发完成阶段性转型需求。

- 转型微服务架构

当系统已经转型到微服务架构时，我们的关注点在于如何更好地维护微服务。在这个阶段，服务定义统一化、无状态服务、服务幂等性、数据最终一致性和版本管理与迁移等都是需要开展的具体工作。

### 8.1.1 调整架构的技术

在本节中，我们将列举有助于调整现有架构的相关技术，包括共享库、共享服务、代码转移、代码冗余、提取新服务和重写服务。

#### 1. 共享库

如果两个服务想要共用一些代码，那么这些代码就可以被提取并放到一个共享库中。共享库体现的是一种代码复用思想，公共代码从某一个服务移到共享库就可以被其他服务所使用。共享库技术的示意图见图 8-2。

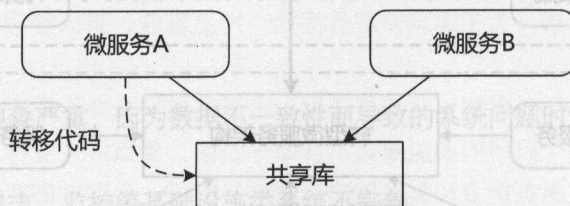


图 8-2 共享库结构示意图

共享库实施过程中的主要挑战在于如何判断某些代码应该移到共享库而不是放在本地服务中。微服务架构实际上并不是非常关注代码复用，因为代码复用会导致服务与服务之间产生新的依赖。微服务架构崇尚服务独立性，过多地把代码提取到共享库可能会引来一些不必要的问题。而且，共享库也会对服务部署结构产生一定影响。

当然，共享库的优势也很明显。本来存在 bug 或不合理设计的代码通过在共享库中的一次修正就能完成所有场景下的代码升级。通过合理管理共享库的版本可以降低代码错误率。

#### 2. 共享服务

共享服务的思路与共享库类似，只不过在共享服务中，被提取的代码并不是放到一个独立



的公共库中,而是直接转移给了另一个服务(见图8-3)。与共享库相比,共享服务的优势在于不用产生新的依赖关系,因为服务与服务之间的交互方式并没有任何改变,而服务内部的任何调整并不会产生架构上的影响。

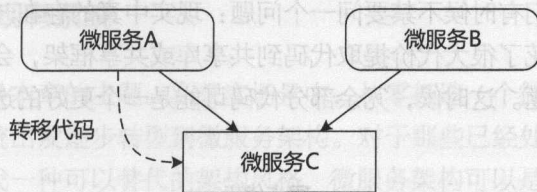


图 8-3 共享服务结构示意图

共享服务是保持服务大小合理性的一种手段,借助共享服务机制,我们可以对规模比较大的服务进行“瘦身”。但是共享服务在不增加服务依赖关系的同时却会转移服务与服务之间的交互需求。当我们把部分代码从一个服务转移到另一个服务时,本来只需要依赖一个服务的场景,就可能变成需要依赖几个服务。

### 3. 代码转移

从一个微服务中抽取一部分代码放到另一个微服务中的做法称为代码转移。通常,我们进行代码转移的主要目的是为了降低两个微服务之间的耦合,提高单个服务的内聚度。当一个微服务需要依赖另一个微服务完成某个功能时,我们认为两者具有一定耦合度,当把两者之间的交互部分代码进行转移之后,这种耦合度就能得到缓解。代码转移示意图如图8-4所示。

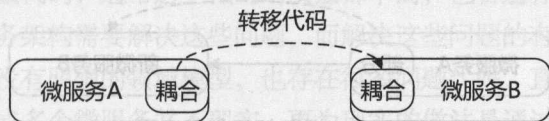


图 8-4 代码转移示意图

代码转移与前面提到的共享库有点类似,但被转移的代码往往到不了高度复用的程度,所以没有必要提取为共享组件。另外,共享库的提取具有技术约束性,即使用共享库的几个微服务一般都需要采用同一种技术实现体系。而代码转移则可以更加灵活,我们完全可以采用另一种技术实现方式重写这些需要转移的代码。微服务一般规模都不大,而需要转移的代码通常只是其中的一小部分,所以基于代码重写的实现方案成本可控。

### 4. 代码冗余

与其把代码转移到另一个微服务,有时候我们也会选择代码冗余的方式降低服务与服务之间的耦合度。在主流的方法论中,普遍认为代码冗余是一项反模式,因为当代码被冗余在两个



地方时，一旦有问题就需要同时修正这两个地方。这是一个架构腐化的危险信号，所以我们一般都尽量避免重复代码的产生。但在微服务架构中，代码冗余有一个非常明显的优势，即两个微服务之间能够保证高度的独立性，从而实现微服务架构所提倡的独立部署。

针对独立部署，我们有时候不禁要问一个问题：现实中真的存在一个完全独立的系统或组件吗？很多时候，我们花了很大代价提取代码到共享库或共享框架，会发现这些共享库或共享框架用起来也有很多问题。这时候，冗余部分代码可能是一个更好的选择。代码冗余的方式见图 8-5。

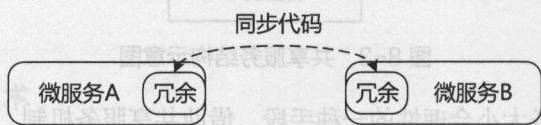


图 8-5 代码冗余示意图

## 5. 提取新服务

当然，我们也可以通过提取部分代码的方式创建一个新服务（见图 8-6）。提取新服务具有与共享服务同样的优点和缺点，但是两者具有不同的初衷。当一个服务的规模逐渐增大，提取新服务的目的在于通过减小服务的规模从而降低服务的维护成本，或者把该服务所承载的一部分职责转移到另一个团队。这时候，这个新服务就不会像共享服务那样被多个服务所共同依赖。

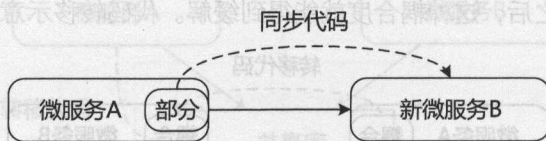


图 8-6 提取新服务示意图

举个例子，在移动医疗系统中，注册流程可能非常复杂，因为需要依赖就诊卡等医疗媒介，也需要区分不同的用户群体，因此我们可以把注册服务拆分成几个服务，分别处理不同的用户注册场景。这种拆分在降低服务复杂度的同时，也会从组织架构上推动团队结构的优化和业务的聚焦。

## 6. 重写服务

最后，如果我们认为一个微服务的结构已经不再合适，我们就只能重写它。相较其他的架构设计方法，微服务架构中的重写并不是一件非常困难的事情，因为微服务的规模较小，同时具备明确的服务契约。

我们有时候会鼓励重写服务，一方面可能来自于技术体系的发展和演进，使用新技术重写



一个老服务会带来更好的发展前景。另一个更重要的方面，重写服务驱使我们再次审视服务背后的领域模型，从而为该领域模型提供一种崭新的、更好的实现。

### 8.1.2 微服务架构与现有系统

向微服务架构转型是本章的主题。在现实场景中，从零打造一个微服务架构的机会很少，更多的是从一个现有系统出发逐步转型到微服务架构。对于那些已经处于架构腐化边缘的现有系统而言，往往正在寻找一种可以替代的架构风格，微服务架构可以是一种潜在的选择。

与构建一个新系统相比，从现有系统向微服务架构转型具有一些显著的特点。

- 业务模型

对于现有系统而言，我们已经具备明确的业务模型。尤其是那些寻找变革的架构，对于业务模型的理解通常很深刻，因为已经经历过业务建模的阵痛期。这点对微服务架构非常重要，也是转型得以实现的基础。

- 领域模型

业务模型不等于领域模型。从领域建模的角度出发，现有系统中存在的业务边界很可能并不满足界限上下文的定义，也没有使用实体、聚合等手段在某个界限上下文内对业务对象进行建模。在这种场景下，向微服务架构转型将面临挑战，因为系统中基于领域的设计方法需要进行调整。

- 遗留代码

现有系统中存在大量代码，通常这些代码的质量都不高，也普遍存在缺少单元测试、部署复杂等典型问题。微服务架构需要解决这些问题，而解决这些问题的有效方法是引入变化。

正因为现有系统中没有明确的领域模型，也存在很多问题代码，直接通过代码拆分的手段将一个完整的系统分割成多个微服务并不现实。更为现实的做法是通过微服务架构来补充和增强现有系统。在本节中，我们将介绍几种在微服务架构与现有系统并存的情况下优化系统架构的方法，包括应用 EAI、应用 UI 集成、应用数据复制以及分离服务。

#### 1. 应用 EAI

EAI (Enterprise Application Integration, 企业应用集成) 是将基于不同平台、使用不同方案建立的异构应用进行集成的一种方法和技术。同时，它也表现为一系列可用于系统集成的架构设计模式<sup>[25]</sup>。

##### (1) EAI 核心组件

EAI 中的核心组件可以分为两大类，一类是消息路由 (Message Routing)，另一类是消息转换 (Message Transformation)。



• 消息路由

消息路由用于将消息传递到某个微服务，我们可以基于消息路由把某些业务请求转发到新的微服务，而不是交由原有系统处理。通过这种方式，我们在开始实施微服务架构的过程中不需要完成整个逻辑，而只需提供部分组件的实现即可。

消息路由需要考虑一次处理单条/多条消息、路由结果面向一个/多个目标、路由是否有状态等问题。围绕上述三个问题所得出的答案，我们加以排列组合可以得到多种路由器的表现形式（见表 8-1），其中有状态的路由器指的就是需要根据消息传递的上下文确定路由结果，通常涉及多个消息，具有较高的复杂性。

表 8-1 路由器总览

路由器	消费消息数	发布消息数	有无状态
内容路由器	1	1	无
过滤器	1	0 或 1	无
接收表	1	0 或 n	无
分解器	1	n	有
聚合器	n	1	有
重排器	n	n	有

内容路由器（Content-based Router）是最简单的路由器，即通过消息的内容决定路由结果。这里的消息内容包括输入消息的消息头属性值、消息体类型以及各种针对消息体内容的自定义的业务规则，通过内容路由器可以产生一对一的路由效果。显然，通过内容路由器，我们可以实现发送消息到某个特定微服务的效果。

接收表（Recipient List Router）面向 1 对多的路由需求，当对同一消息进行路由时，特定场景下可能会满足多种路由条件从而产生多个路由结果（见图 8-7）。接收表在现有系统与微服务系统并存的情况下也有一定的应用场景，我们可以在不改造现有系统的前提下，透明地将同一个消息分别发送给现有系统和微服务系统，并通过微服务系统进行最后结果的处理，最终慢慢替换掉现有系统，这实际上也可以理解为绞杀者模式的一种具体应用。

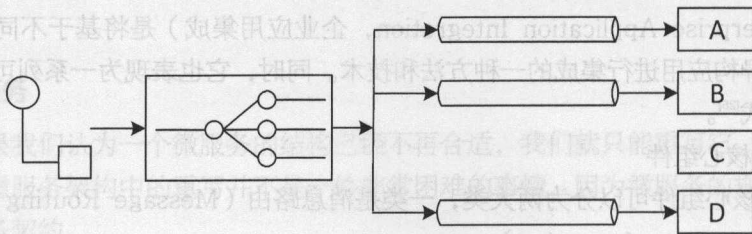


图 8-7 路由表示意图



过滤器 (Filter) 的目的是决定是否将消息流转到下一个环节, 如果满足一定过滤条件, 则该消息将不会产生任何路由结果, 过滤条件同样可以包括复杂的业务流程性内容。对于微服务所不能支持的某些消息而言, 过滤器可以帮助我们实现消息过滤。

分解器 (Splitter) 的典型应用场景是消息包含多个元素, 而每个元素处理方式不同, 这时候我们可以把原始消息分解成多个消息, 并通过复制关联标识符 (CorrelationId) 等公共属性的方式实现分解后消息的关联。聚合器 (Aggregator) 往往和分解器一起使用, 是分解的逆过程, 将独立而又相关的消息组织成整体进行处理。聚合器的实现具有典型的状态性, 因为独立消息的个数、到达顺序、相关性等因素都依赖于消息传递上下文。有时候, 分解之后的独立消息并不一定能在有限的时间间隔之内都到达聚合器, 而聚合器也不可能无限等待, 这就需要明确聚合完成策略。常见的聚合策略有等待所有、第一个最好、超时等, 图 8-8 所示的就很可能是一个等待所有的聚合完成策略。分解器和聚合器组合与 Hadoop 中 Map/Reduce 算法有异曲同工之妙, 对于数量较大的计算可以通过路由的策略并行执行, 这也是解决类似问题的一种思路。分解器和聚合器的应用取决于场景, 在微服务架构中可以根据需要灵活应用这两个组件的组合。

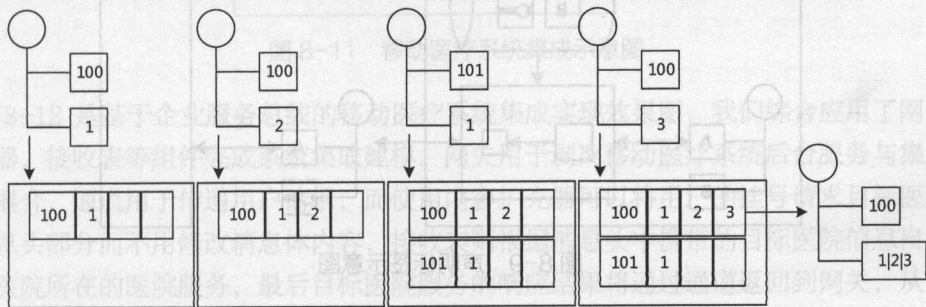


图 8-8 聚合完成策略示意图

几种消息路由实现可以相互组合形成组合处理器, 例如, 可以使用分解器+路由器+聚合器的组合方式实现非常复杂的现有系统和微服务架构之间的服务整合需求。

### • 消息转换

消息转换器 (Transformer) 解决服务与服务之间数据如何在异构系统之间进行适配的问题。现有系统与微服务之间经常需要异构系统之间的交互和集成, 通过转换器可以消除异构系统之间由于数据格式所导致的依赖。最基本的转换思路就是通过一种自定义转换机制进行两种数据结构之间的映射, 但有些场景下我们也需要实现基本数据结构转换, 并对输入的数据结构进行内容的扩充和过滤。

内容扩充器 (Content Enricher) 就是往消息中扩充新的数据, 扩充的数据来源可以来自计算、外部环境和第三方其他系统, 扩充的对象可以是消息的消息头也可以是消息体, 所以内容扩充器一般可以分成消息头扩充器 (Header Enricher) 和消息体扩充器 (Payload



Enricher)。内容过滤器 (Content Filter) 是内容扩充器的对称操作, 内容过滤的目的是去除消息中的某一部分而不是在消息传递过程中过滤掉该消息。

声明标签 (Claim Check) 的作用在于减少传输量和隐藏敏感信息, 体现的是一种数据直接传输和间接存储相结合的思想。如果存在一个数据中央仓库保存着数量大或安全性要求高的信息时, 在消息传递过程中我们就可以做到把数据的业务键、消息 Id 或任何唯一性 Id 作为消息键进行传递, 消息的接收方可以根据该消息反向从数据中央仓库中获取目标数据。声明标签的结构示意图见图 8-9, 我们可以通过在现有系统和微服务之间传递消息键来实现两者之间有限信息的传递, 而微服务在拿到这个消息键之后可以根据需要开展后续业务。

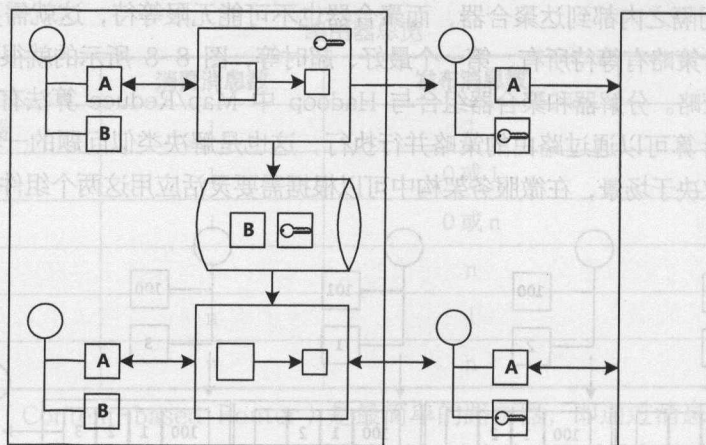


图 8-9 声明标签示意图

## (2) EAI 应用示例

图 8-10 展示了一个简单场景, 消息路由器接收请求并把消息转发到某个微服务或现有系统。在这个场景中, 我们可能已经使用绞杀者模式对系统中的部分功能进行了微服务改造, 同时保留了部分原有功能。当业务请求到达系统, 可以通过业务请求的属性进行服务路由, 分别转向已改造完成的微服务以及尚待改造的现有系统。

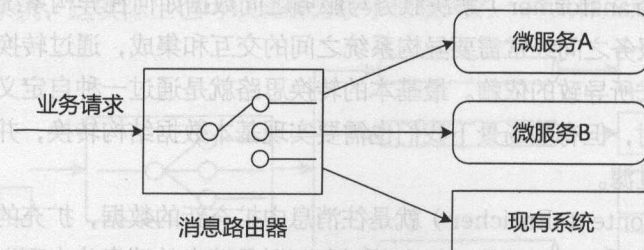


图 8-10 使用消息路由器集成现有系统



以移动医疗系统为例，针对预约挂号这个业务，可能不同的医院有不同的逻辑，所以现有系统中会同时存在若干套预约挂号流程。在向微服务架构转型的过程中，就可以采用图 8-10 中的路由机制实现新老流程之间的平滑过渡。

我们再来看一个例子。图 8-11 展示的还是移动医疗系统预约挂号模块中获取挂号信息功能的业务场景，用户通过手机 APP 连接到移动医疗系统中的挂号服务，而挂号服务则根据用户的挂号选择导向目标医院。针对具体某家医院，挂号服务需要通过系统集成的手段获取该医院的挂号详细信息并返回给用户。这个过程中，由于涉及与外部各个医院之间的信息传递和交互，我们就可以通过 EAI 中的各个组件对其建模并实现。

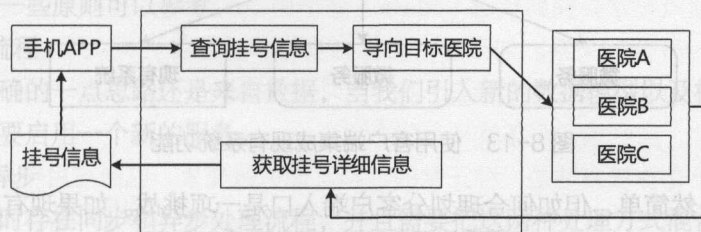


图 8-11 移动医疗系统集成示意图

图 8-12 是基于企业服务总线的移动医疗系统集成实现效果图，我们综合应用了网关、内容扩充器、接收表等组件完成系统集成建模。网关用于剥离移动医疗系统后台服务与集成服务之间的耦合，通道用于传递用户请求，而使用内容扩充器可以将用户的挂号请求目标医院信息放到消息头部分而不用修改消息体内容，接收表则根据消息头中携带的目标医院信息自动路由到目标医院所在的医院服务，最后目标医院服务的响应结果将通过通道返回到网关，从而实现系统集成的闭环。

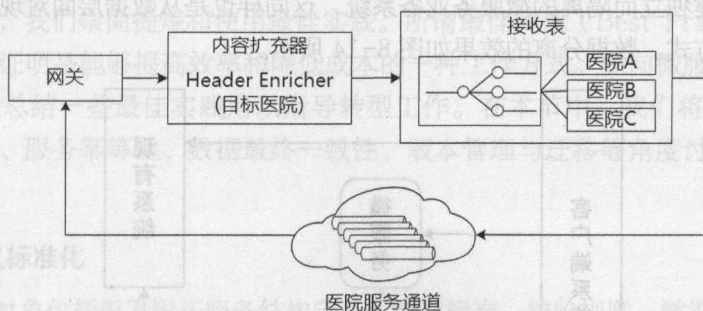


图 8-12 基于 EAI 的移动医疗系统集成实现效果图

## 2. 应用客户端集成

相比通过各种企业应用集成模式来实现微服务架构与现有系统之间的协作，客户端集成的



方式简单而有效。我们在 3.2.7 节中已经深入讨论过客户端集成的各项技术，以 BackEnd For FrontEnd 服务器为例，我们可以通过该服务器做一层转发操作，将系统的一部分请求转发到微服务，而另一部分请求则仍然流向现有系统。图 8-13 展示了这种方式的基本结构。

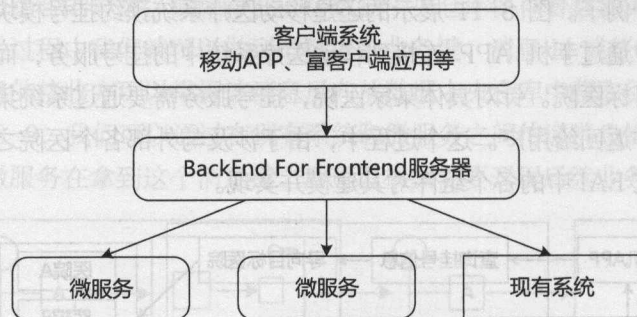


图 8-13 使用客户端集成现有系统功能

客户端集成虽然简单，但如何合理划分客户端入口是一项挑战。如果现有系统的各个客户端入口规划比较合理，简单通过入口划分就能界定业务边界，客户端集成是一项很好的请求转发机制。当我们把部分业务通过微服务进行实现之后，新的请求就可以完全脱离现有系统转而是由新的微服务处理。但在很多现有系统中，客户端入口并不是非常独立，可以针对那些边界比较清楚的入口采用客户端集成方式，其余入口则结合前面的 EAI 方式进行综合处理。

### 3. 应用数据分离

随着引入微服务之后部分业务功能的解耦，系统中已经存在一些微服务能够独立提供业务功能。这个时候我们就可以逐渐考虑开展数据分离工作。通过从现有系统中剥离部分业务相关的数据，尽量构建独立而隔离的微服务业务系统，这同样也是从数据层面对现有系统进行“绞杀”的一种有效方式。数据分离的效果如图 8-14 所示。

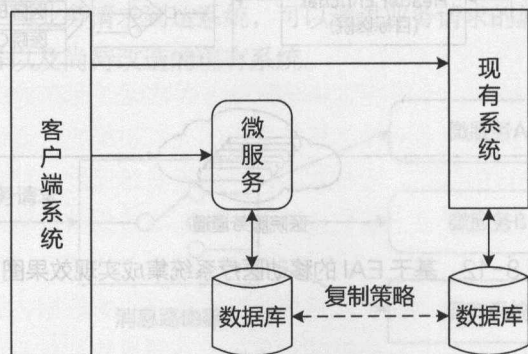


图 8-14 数据分离效果图



数据分离对于一部分业务是可以直接应用的，但对业务逻辑较为复杂的现有系统而言，数据分离需要花费很大代价，这点我们已经在 2.4.2 节中有所介绍。所以在向微服务架构转型的过渡阶段，数据复制也是需要采用的一种策略。在图 8-14 中，我们可以在现有系统和微服务之间采用离线批量策略开展数据同步，如果数据同步实时性要求较高，那就只能采用服务接口的方式集成。关于数据复制，我们也在 3.2.6 节中做了详细讨论。

#### 4. 应用服务分离

这里服务分离的含义在于我们是否要启动一个新的服务还是继续沿用已经存在的老服务。关于这个问题有一些原则可以参考。

- 数据和流程

首先比较明确的一点思路还是来自数据，当我们引入新的数据模型以及相应的数据处理流程时，一般都需要启用一个新的服务。

- 2 同步和异步

当系统中同时存在同步和异步处理流程，并且需要把这两种处理方式混合起来时，我们需要考虑是否应该引入新的服务以简化这种处理流程。

- 服务切面

在一个现存服务中，当我们发现其具有不同的切面（Aspect）时，例如，某些通用机制不断在业务代码中重复出现，或者某个功能有多个不同的实现场景时，我们可以从切面的角度出发，分析是否需要将这些切面抽取出来形成新服务。

### 8.1.3 微服务实施最佳实践

在软件行业，我们崇尚提炼和使用最佳实践。所谓最佳实践（Best Practice）是基于很多人的验证、被证明是能够提高效率和降低成本的一种工作方式。在向微服务架构转型过程中，我们也可以总结一些最佳实践用以指导转型工作。在本节中，我们将从服务定义标准化、无状态服务、服务幂等性、数据最终一致性、版本管理与迁移等角度讨论微服务实施的最佳实践。

#### 1. 服务定义标准化

服务定义的对象包括但不限于服务结构定义、数据标准、校验规则、数据转换规则等，对于这些对象，标准化有两层含义。

- 全局统一

全局统一代表服务的定义在整个产品开发、测试、交付过程中都采用同一种表现形式和表现媒介，这为服务本身在整个生命周期中的发展和演进提供标准，涉及技术层面的系统集成，



也涉及组织层面的团队协作。我们在 2.2.2 节中讨论了服务的统一表现形式。

- 机器可读

机器可读的目的在于为服务的自动化提供前提条件。这里的自动化可以包含开发过程的自动化，如自动根据语义生成服务定义的 API 文档；也包括测试过程的自动化，如 7.1.3 节中讨论的消费者驱动契约测试就有这方面的需求；同时也包括交付过程的自动化，通过自动识别服务定义，可以在持续集成阶段横向加入各种发布前的前置校验。

## 2. 无状态服务

在软件设计领域经常会提到状态（State）这个词，那么什么是服务之间的状态？状态本质上体现的还是一种数据关系。如果一个数据需要在多个服务之间共享才能完成一项业务功能，那么这项业务功能就被称为有状态。基于这项业务功能所设计和实现的一系列服务之间就形成了一种状态性，这一系列服务就是有状态服务。

当然，业务上存在状态性是一种正常的、不可避免的现象。电商应用中最典型的 Session 就是状态性的一种表现，用户的登录、购物等环节都需要依赖 Session 信息，这些环节背后的不同微服务就形成了一种 Session 共享的场景。但在技术实现上，有状态服务显然会促使服务与服务产生耦合，从而增加了服务体系的复杂度。同时，考虑到系统的可伸缩性，我们也要实现服务与服务之间的无状态化处理。

实现无状态服务的手段也有很多，例如，状态共享，即把状态保存在某一个数据存储媒介中，常见的分布式缓存或持久化数据库都是可以采用的实现技术。另外，我们可以采用客户端持有状态的方式解耦服务与服务之间的状态依赖关系，在每次服务调用过程中，客户端将状态信息传递给服务方，7.4.3 节讨论的客户端 Token 实际上就是这种策略的具体体现。当然，目前市面上的一些中间件也能为我们提供状态的复制（Replication）或粘滞（Sticky）机制，前者利用服务器之间的状态同步实现所有请求都能获取相同的状态信息，从而提供一个对等网络，而后者则使用诸如 IP 等请求信息固定客户端与服务器端的映射关系，从而消除状态所带来的副作用。

复制或粘滞机制在 Nginx、HAProxy 等主流的应用服务器和代理服务器中都有所体现。例如，Nginx 就提供了专门用于粘滞的 Sticky 模块，它是基于 Cookie 的一种 Nginx 的负载均衡解决方案，通过分发和识别 Cookie，确保一个客户端的请求落在同一台服务器上，Cookie 的默认标识名为“route”。图 8-15 展示的就是 Nginx 中 Sticky 模块的工作流程图。客户端首次发起访问请求，Nginx 接收后发现请求头没有 Cookie，则以轮询方式将请求分发给后端服务器。然后后端服务器处理完请求，将响应数据返回给 Nginx，此时 Nginx 生成带“route”标识的 Cookie，返回给客户端。“route”标识的值与后端服务器对应，可能是明文，也可能是 md5、sha1 等 Hash 值。客户端接收请求，并保存带“route”标识的 Cookie。



这样当客户端下一次发送请求时，会带上“route”标识，Nginx 根据接收到的 Cookie 中的“route”值，转发给对应的后端服务器。

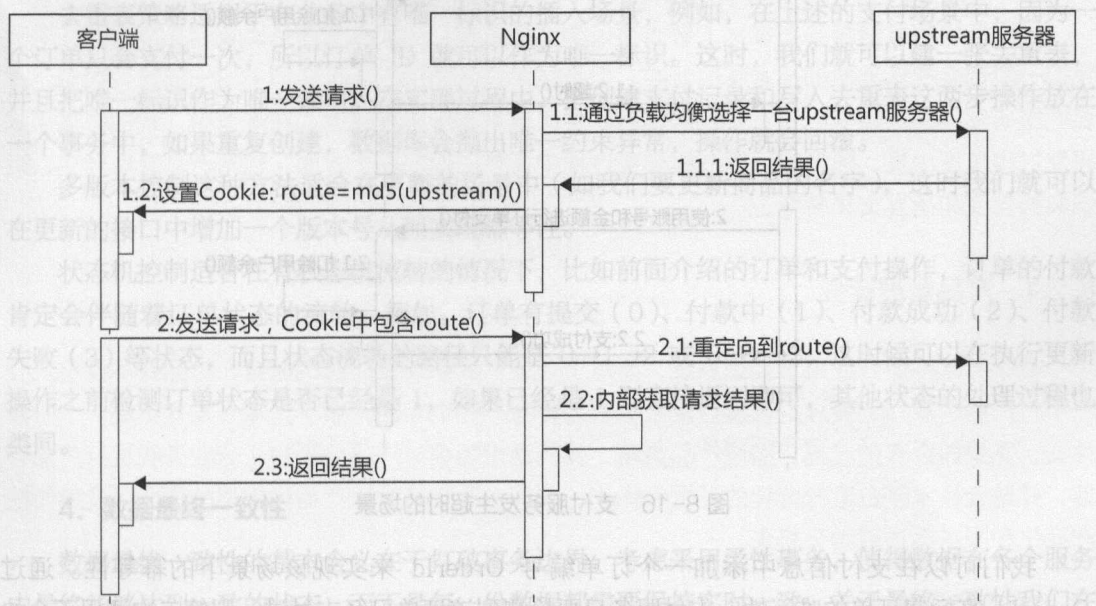


图 8-15 Nginx Sticky 模块的工作流程图

### 3. 服务幂等性

所谓幂等性 (Idempotency) 指的是一次和多次请求某一个资源应该具有相同的作用，站在接口调用的角度意味着接口可重复调用，在调用方多次调用的情况下，接口最终得到的结果应该保持一致。显然，有些接口可以天然地实现幂等性，如查询接口。但是对于增删改操作而言，问题就会变得比较复杂，尤其是在分布式环境下的服务交互和协作场景。

我们通过一个例子来解释为什么在微服务架构中需要确保幂等性。试想订单交易业务中存在两个微服务，一个是订单服务，另一个是支付服务。在某一个具体场景下，发生了如图 8-16 所示的交互过程。支付服务已经扣款，但是订单服务因为网络原因，没有获取到确切的结果，因此订单服务需要重试。

从图 8-16 可见，支付服务并没有做到接口的幂等性，订单服务第一次调用和第二次调用，用户分别被扣了两次钱，不符合幂等性原则。该场景下的幂等性指的是同一个订单，无论调用了多少次，用户都只会被扣款一次。



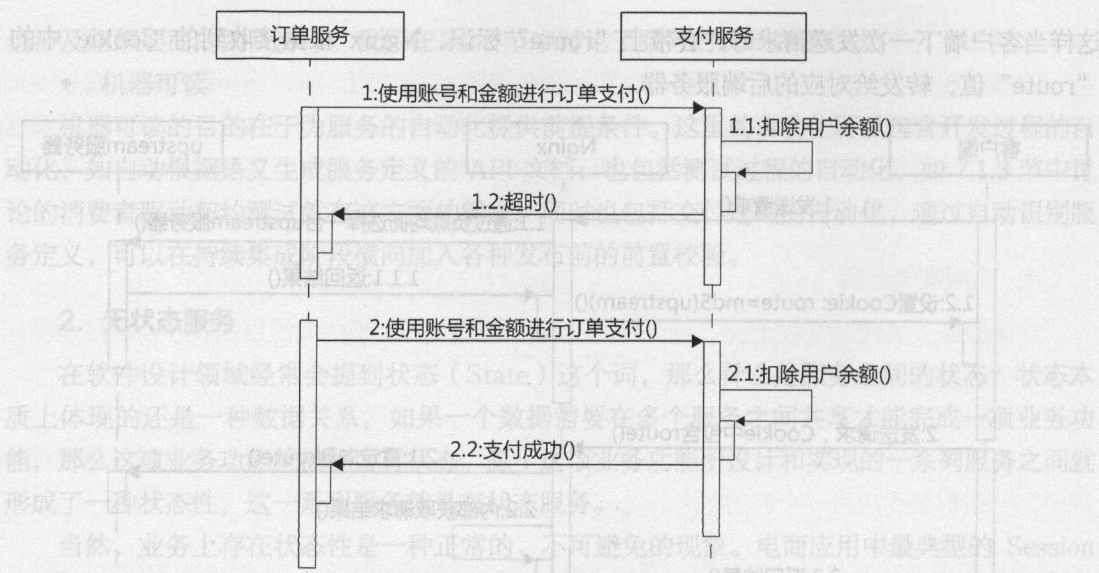


图 8-16 支付服务发生超时的场景

我们可以在支付信息中添加一个订单编号 OrderId 来实现该场景下的幂等性。通过 OrderId 来标定订单的唯一性，支付服务只要检测到该订单已经支付过，则第二次调用不会扣款而会直接返回结果（见图 8-17）。

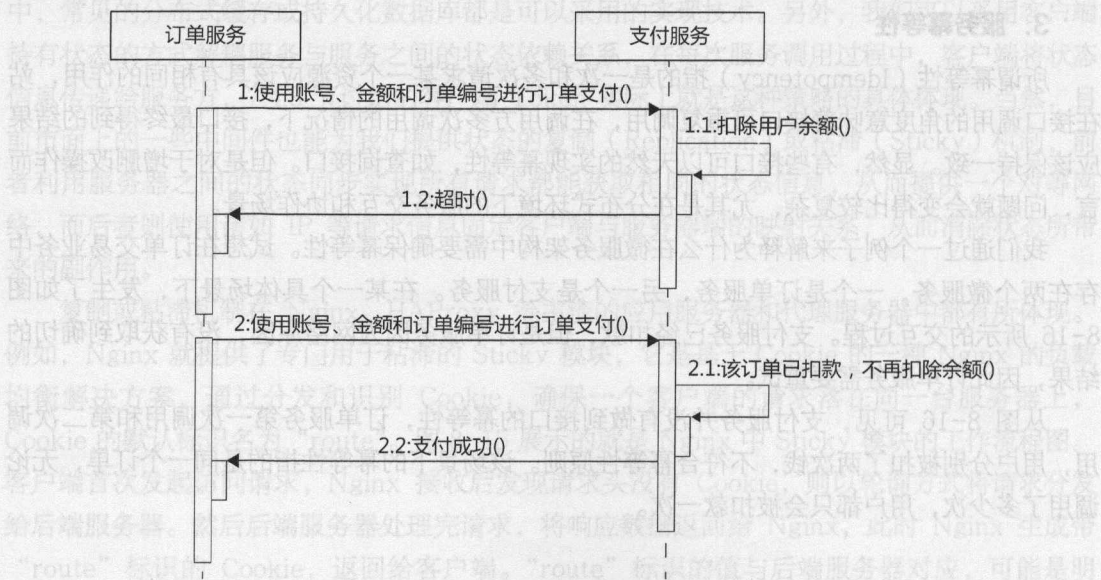


图 8-17 通过唯一订单编号实现支付幂等性



以上关于订单和支付场景的讨论，我们得出的是使用全局唯一 ID 的方式来实现幂等性。除了这种方式之外，我们还可以采用的策略包括去重表、多版本控制、状态机控制等。

去重表策略适用于在业务中有唯一标识的插入场景，例如，在上述的支付场景中，因为一个订单只会支付一次，所以订单 ID 就可以作为唯一标识。这时，我们就可以建一张去重表，并且把唯一标识作为唯一索引。在实现过程中，把创建支付记录和写入去重表这两步操作放在一个事务中，如果重复创建，数据库会抛出唯一约束异常，操作就会回滚。

多版本控制这种方法适合在更新的场景中（如我们要更新商品的名字），这时我们就可以在更新的接口中增加一个版本号从而实现幂等性。

状态机控制适合在有状态机流转的情况下，比如前面介绍的订单和支付操作，订单的付款肯定会伴随着订单状态的流转。例如，订单有提交（0）、付款中（1）、付款成功（2）、付款失败（3）等状态，而且状态流转的路径只能是 0→1→2 或 0→1→3，这时候可以在执行更新操作之前检测订单状态是否已经是 1，如果已经是 1 则直接返回即可，其他状态的处理过程也类似。

#### 4. 数据最终一致性

数据最终一致性的基本含义在于打破事务边界，考虑采用柔性事务，使得数据在各个服务中最终能够达到一致的状态，而不是每一份数据都需要保持实时一致。关于最终一致性我们在 5.2 节已经做了深入分析并给出了几种常见的实现方案，包括可靠事件模式、补偿模式、Sagas 长事务模式、TCC 模式、最大努力通知模式以及最基本的人工干预模式。

当然，对于那些必须追求强一致性的业务需求，我们需要转变思路看看是否可以避免出现分布式事务的场景。在设计微服务时，如果能将所有的业务功能都限制在一个服务之内，也就不需要考虑分布式的复杂应用场景，这也是解决这类问题的一种思路。

#### 5. 版本管理与迁移

版本控制适用于任何 API，对微服务也一样。如果有某些改动打破了 API 的原始定义，那么就应当针对该改动单独发布另外一个版本。因为不论是公共接口还是其他内部服务使用的接口，我们不清楚具体谁在使用这些接口，因此必须要保证向下兼容，或者至少要给使用这些接口的用户足够的时间去做调整。

在微服务架构实施过程中，可以预见复杂度会随着服务数量的增加而增加。我们已经在 3.2.7 节介绍 BackEnd For FrontEnd 服务器时提到过版本管理和迁移的基本实施方案。制订类似的一个版本控制策略可以使消费者轻松迁移，并且服务提供者可以透明地部署变更而不影响其他任何人。但同时，我们也要限制生产环境中并行的服务版本数量，在完成版本升级之后，确保老版本服务得到合理的治理。



## 8.2 微服务架构与研发过程转变

微服务架构改造需要在技术上做出很多尝试和调整,这实际上还是比较容易把握的,毕竟我们一般都会挑选成熟的技术解决方案。但是技术只是微服务架构转型中的一部分,我们还需要把握研发过程的转变,这点实现起来有时候比技术本身难度更大。在本节中,我们将重点从研发过程角度出发,梳理在研发过程转变过程中需要把握的各个方面,包括产品管理转变、组织架构转变和研发文化转变。

### 8.2.1 产品管理转变

微服务架构适合应用于大型系统,而对于大型、复杂的业务结构而言,产品并不仅仅只有一个,产品的规划、设计、开发、发布等都需要有一定的策略。由于产品开发并不是一个阶段性的、完成即丢弃的过程,围绕产品策略,我们通常需要对产品分平台、分业务线进行产品管理,并结合技术平台进行持续的开发和优化。本节中我们将讨论产品化框架的构成以及如何把产品化和服务化进程结合起来。

#### 1. 产品化框架

从产品开发过程角度出发,产品构想的实现需要两个主要步骤:分解和分层。分解和分层的背后都体现了一种产品开发的理念,即组合理念。对于一个大型事物而言,可以通过组合若干个小型事物的方式进行构建。分解和分层的目标就是明确产品内部构成的边界,将产品合理划分成一个集合,从而为产品构建提供基础。组合理念表现为图 8-18 中的示意图,我们可以看到,业务分层、技术平台和业务模块相互组合可以形成产品的基本形态。

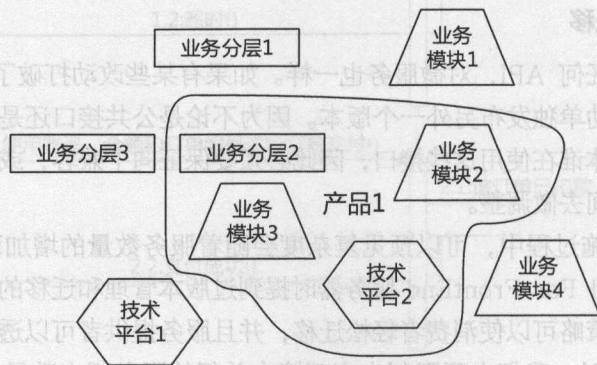


图 8-18 组合理念示意图



对于产品开发，分解指的是将产品构想中的业务和技术部分内容进行分离。业务和技术分离的优势很明显，因为业务和技术本身就应该独立发展，独立的技术组件能够为产品提供复用支持。而分层则更多关注于业务自身的层次结构，业务也需要区分基础业务和普通业务，基础业务作为业务组件同样为业务组合提供了业务复用性。

基于产品的分解和分层步骤，我们得到了如图 8-19 所示的产品开发框架。产品化框架体现的是一种组合理念，并将产品开发分成产品线、产品平台和技术平台三个组成部分。

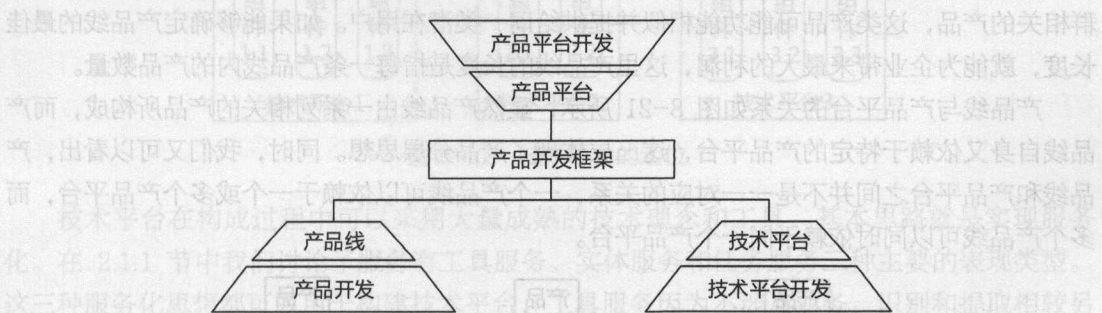


图 8-19 产品开发框架

### (1) 产品平台

产品平台是共同业务要素的一个集合，指的是一系列产品实施过程中采用的基础业务。从产品的角度上看，这些共同业务要素对于最终可发布的目标产品而言，不必是完整无缺的内容。但这些要素却是目标产品的必要构成部分，图 8-20 展示了产品平台在产品演进过程中的定位。我们可以看到每个产品都会依赖产品平台中一个或多个业务基础组件，而在现有产品上添加新的组件就可能构建新的产品，例如图 8-20 中的产品 A 和产品 A1。

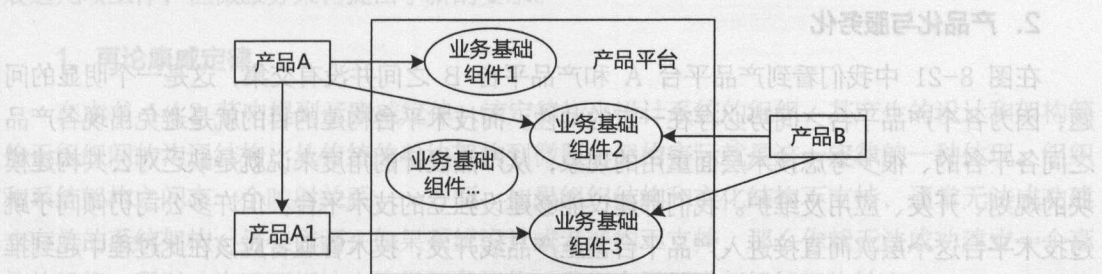


图 8-20 产品平台的定位

产品平台设计的根本要素在于确定产品平台的边界和提供访问产品平台的入口。产品平台边界的确立在于合理划分模块和组件的粒度，而其所提供的入口则将以平台级别的方式供外部产品或产品线进行业务集成。

一个产品平台的构成要素，包括平台的访问入口以及平台背后的各个模块和组件，这些模



块和组件在平台内部存在有序且有限的依赖关系。我们可以看到，产品线可以通过产品平台所提供的统一入口获取产品功能，同时，产品线有时候也可以直接访问产品平台中的某一个共享模块或组件，以便整合其功能。作为一项最佳实践，可以将所有的访问前再加一层门面（Facade），以降低外部使用者与产品平台之间的依赖。

## （2）产品线

对于产品开发而言，一般都会存在产品线（Product Line）的概念。所谓产品线，是指一群相关的产品，这类产品可能功能相似并提供给同一类潜在用户。如果能够确定产品线的最佳长度，就能为企业带来最大的利润，这里产品线的长度是指每一条产品线内的产品数量。

产品线与产品平台的关系如图 8-21 所示，显然产品线由一系列相关的产品所构成，而产品线自身又依赖于特定的产品平台，这三层体现了产品分层思想。同时，我们又可以看出，产品线和产品平台之间并不是一一对应的关系，一个产品线可以依赖于一个或多个产品平台，而多个产品线可以同时依赖于同一个产品平台。

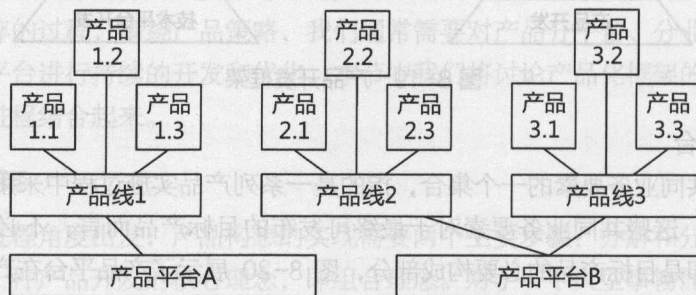


图 8-21 产品线与产品平台的关系

## 2. 产品化与服务化

在图 8-21 中我们看到产品平台 A 和产品平台 B 之间并没有交集，这是一个明显的问题，因为各个产品平台之间势必存在一定的共性。而技术平台构建的目的就是避免出现各产品之间各干各的、很少考虑技术层面重用的现象，从产品设计的角度来说就是缺乏对公共构建模块的规划、开发、应用及维护。我们都建议能够建设独立的技术平台，但许多公司仍倾向于跳过技术平台这个层次而直接进入产品平台甚至产品线开发，技术管理者应该在此过程中起到推动作用。

技术平台的定位非常明确，即为产品平台提供技术支持（见图 8-22）。技术平台的目的在于提供核心技术服务，这些技术服务具有高度复用性和独立性，并实现即插即用的使用效果。技术平台中包含了一系列的技术组件，我们也可以在图 8-22 中进一步明确，产品平台与这些技术组件之间也表现为多对多的对应关系。



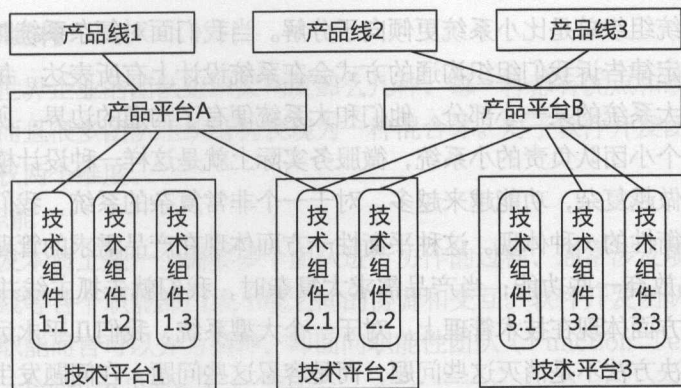


图 8-22 技术平台的定位

技术平台在构成过程中可以采用大量成熟的技术理念和工具，基本思路就是实现服务化。在 2.1.1 节中我们讨论了服务有工具服务、实体服务和任务服务三种主要的表现类型。这三种服务化思想都可以用于构建技术平台，工具服务因为不涉及业务，识别和提取相较另外两种服务而言比较简单。实体服务关注于数据，任务服务关注于业务组合，都需要根据业务本身做抽象。

## 8.2.2 组织架构转变

对于软件开发而言，从团队组织结构出发，对团队所需工作进行分析和设计，并基于工作需要组建合适的团队构成技术管理者的几项主要工作事宜。在推行微服务架构中，同样需要开展这几项工作，但微服务架构提出了新的要求。

### 1. 再论康威定律

在本书 1.4.2 节中提到了康威定律，该定律指出设计系统的组织，其产生的设计和架构等价于组织间的沟通结构。从传统的单块架构到微服务架构实际就是这一定律的一种体现。组织和系统架构之间有一个映射关系，一方面，如果组织结构和文化结构不支持，通常无法成功建立有效的系统架构；另一方面，如果系统设计或者架构不支持，那么你就无法成功建立一个高效的组织。所以，为了更好地实施微服务架构，我们也需要进行组织架构转变。

本节在 1.4.2 节的基础上进一步讨论康威定律，引用 Mike Amundsen 在《远距离条件下的康威定律：分布式世界中实现团队构建》<sup>[26]</sup>一文的一些观点。Mike 认为康威定律在以下几个方面影响着组织架构的发展。

首先，组织沟通方式决定系统设计。组织的沟通和系统设计之间的紧密联系，在很多别的领域有类似的阐述。对于复杂的系统，解决好人与人的沟通问题，才能有一个好的系统设计。



其次，大的系统组织总是比小系统更倾向于分解。当我们面对复杂系统时，基本的思路就是分而治之。康威定律告诉我们组织沟通的方式会在系统设计上有所表达，每个管理者都被赋予一定的职责去做大系统的某一小部分，他们和大系统便有了沟通的边界，所以大的系统也会因此被拆分成一个个小团队负责的小系统，微服务实际上就是这样一种设计模式。

再次，系统越做越复杂，功能越来越多，对于一个非常复杂的系统，我们永远无法考虑周全，所以这也是平衡性的一种体现。这种平衡性一方面体现在产品需求的管理上。当产品需求太多时，我们可以放弃一些功能；当产品需求太复杂时，我们就先抓主线并适当忽略一些细节。平衡性的另一方面体现在技术管理上。对于一个大型系统，我们几乎永远不可能找到并修复所有的问题，解决方法不是消灭这些问题，而是容忍这些问题，在问题发生时，让该问题能够自动修复。例如，在由微服务组成的系统中，每一个微服务都可能出现问題，但我们只要有足够的冗余和备份就可以保证系统的高可用性。

最后，线型系统和线型组织架构间有潜在的异质同态特性。直白地说，你想要什么样的系统，就搭建什么样的团队。如果一个团队在组织上交成交互体验团队、Java 后台开发团队、DBA 团队和运维团队，那么系统就会长成图 8-23 中的样子。

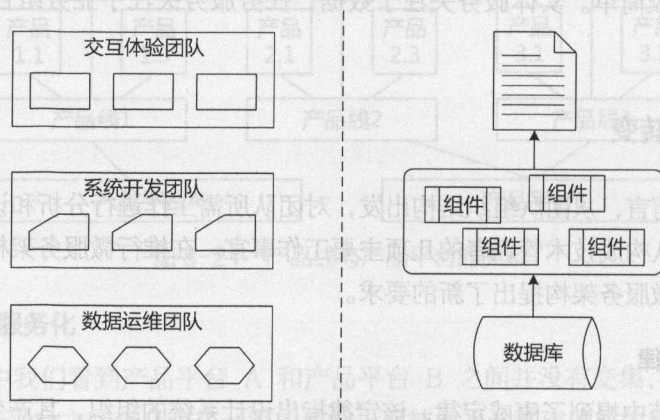


图 8-23 团队与系统之间的关系

相反，如果你的系统是按照业务边界划分的，大家按照一个业务目标去把自己的模块做成小系统、小产品，大系统就会演变成微服务的架构。在一个团队内具备全栈能力，让团队自治，可以将沟通的成本维持在系统内部，每个子系统就会更加内聚，彼此的依赖耦合能变弱，跨系统的沟通成本也就能降低。

康威定律给了我们很多启示，其中最重要的一条就是最好按业务来划分团队，这样能让团队自然地自治内聚，明确的业务边界会减少与外部的沟通成本。接下去我们顺着康威定律的启示并结合微服务架构的实施过程来讨论团队的组织结构。



## 2. 团队组织结构

实际上，全世界企业的团队组织架构就那么几种，每一种都有优点和缺点，没有一种组织结构是完美的，而且很多团队组织结构表现为一种混合体。对于软件开发团队而言，通常我们关注于职能和职权两个维度。

### （1）按照职能

软件项目研发本质上是一个需要多个团队进行协作的过程，通常涉及项目线、产品线、技术线、质量保证线等各个职能部门或小组之间的协调和交互。软件开发团队的团队组织结构根据是否从事单一职能而言可以分为两种，即面向职能性团队（Function Team）和跨职能团队（也叫特征团队，Feature Team）。

#### • 职能型团队

职能型组织结构亦称 U 型组织，即以工作方法和技能作为部门划分的依据。在软件研发过程中，产品、项目、开发、测试、运维等各个角色都可以形成独立的职能团队。因为业务活动都需要有专门的知识 and 能力。通过将专业技能紧密联系的业务活动归类组合到一个团队内部，可以更有效地开发和使用技能，提高工作的效率。

职能型团队的最大问题在于没有一个直接对项目或产品负责的强有力的权力中心或个人，各个职能团队之间因为没有形成统一的目标导向，其协调过程十分困难。

#### • 跨职能团队

而对于跨职能团队而言，在团队内部应该包括产品、开发、测试、运维等各种角色，能够完成某个产品或服务的整个生命周期，组合职能型团队与跨职能团队的组织结构见图 8-24。

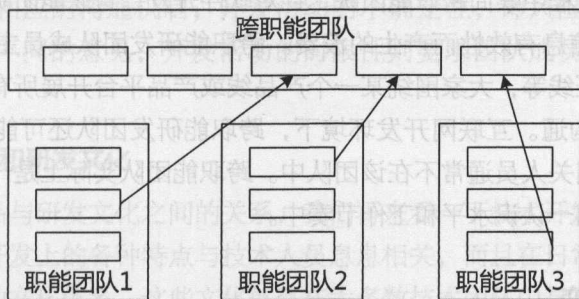


图 8-24 组合职能型团队与跨职能团队组织结构

跨职能团队是一种有效的团队管理方式，它能使团队内（甚至团队之间）不同领域的成员之间交换信息，激发产生新的观点，解决面临的问题，协调复杂的项目。但是跨职能团队在形成的早期阶段需要耗费大量的时间，因为团队成员需要学会处理复杂多样的工作任务。在成员之间，尤其是那些背景、经历和观点不同的成员之间，建立起信任并能真正地合作也需要一定的时间。



## （2）按照职权

从项目管理角度而言，团队组织结构一般都会使用矩阵型结构，而矩阵型结构根据项目经理的职权大小又可以分为强矩阵、弱矩阵和平衡矩阵团队。这三种矩阵团队组织的特点参考表 8-2<sup>[27]</sup>。

表 8-2

三种矩阵式组织结构

	弱矩阵	平衡矩阵	强矩阵
项目经理的职权	小	小到中	中到大
项目经理的角色	兼职	全职	全职
项目预算控制权	职能经理	混合	项目经理
可用的资源	少	小到中	中到大

弱矩阵组织结构基本保留职能组织结构的大部分主要特征，为了更好地实施项目，一般会建立相应的项目管理虚拟团队。

平衡矩阵组织结构是对弱矩阵组织结构的改进，为强化对项目的管理，在项目管理团队内，从职能部门参与本项目活动的成员中任命一名项目经理。项目经理被赋予一定的权力，对项目整体与项目目标负责。

强矩阵组织结构具有项目的线性组织结构的主要特征。强矩阵组织结构在系统原有的职能组织结构的基础上，由系统的最高领导任命对项目全权负责的项目经理，项目经理直接向最高领导负责。

微服务架构实施过程中崇尚跨职能团队。针对软件开发，跨职能团队可以用于消除在信息传递过程中为了确保信息有效性而产生的浪费。跨职能研发团队成员主要包括项目线、产品线、技术线、质量保证线等，大家围绕某一个产品线或产品平台开展所有工作，确保坐在一起并能够实时、面对面沟通。互联网开发环境下，跨职能研发团队还可能包括运营、客服等角色，但销售、市场等相关人员通常不在该团队中。跨职能团队实际上是一种强矩阵的团队组织结构，所有人保持在同一认识水平和工作节奏中。

## 8.2.3 研发文化转变

对于研发团队而言，引入变化的目的是为了打破个人和组织的思维惯性，从而形成新的研发文化。研发文化的建立需要因地制宜，一般取决于两个重要条件，即团队所处的不同发展时期以及团队所承担的产品开发特点。同时，技术开发本身具有的一些特点也是研发文化的一部分。



1. 团队生命周期和产品研发特征

根据布鲁斯·塔克曼（Bruce Tuckman）的团队发展阶段模型<sup>[28]</sup>，一个团队的发展一般需要经历五个阶段，即组建阶段（Forming）、震荡阶段（Storming）、规范阶段（Norming）、执行阶段（Performing）和休整阶段（Adjourning），并且这五个阶段都是必需的、不可逾越的，团队在成长、迎接挑战、处理问题、发现方案、规划、处置结果等一系列经历过程中必然要经过上述五个阶段。

同时，产品的研发并不一定是一个从无到有的过程，正如我们在前面 8.2.1 节中提到的产品化策略与平台一样，产品的开发也有几种不同的表现形式，而不同的产品研发类型也决定了产品的开发团队类型。表 8-3 列举了三种比较典型的产品开发类型以及所对应的团队特征。

表 8-3 新产品开发类型

产品开发类型	团队类型	团队特点	管理要素
新平台产品开发	独立的专项团队	独立于日常开发	保持团队高度自治
完整现有产品线	跨部门临时团队	开发工作与其他日常工作并行	加强部门协调机制
产品技术改进	技术改进团队	涉及范围较小，组建方式灵活	把握项目运行时机

研发团队文化是企业整体文化的组成部分，因此它具有企业文化的共性。但根据不同的产品研发特征，它又有特性和自身要求。例如，开发活动的创新性就要求鼓励团队成员原创性的工作，但鼓励创新也必须建立在团队合作的基础上；开发活动的协同性要求鼓励研发团队成員随时随地交流，要有相应的沟通机制；开发活动的不确定性，即风险性要求研发团队重视工作细节和团队成员不同的意见；开发活动的时限性则要求团队成员要有强烈的时间观念和责任意识。

2. 技术开发特点和研发文化

讨论完团队、产品与研发文化之间的关系，我们再来看一下技术开发本身的特点与研发文化之间的关系。技术开发上的各种特点与技术人员息息相关，而且在日常开发过程中也不知不觉就形成了一些特定的文化体系。这些文化体系在大多数技术团队中具有代表性，包括工具文化、过程文化、代码文化、数据文化和工程实践文化。

• 工具

对于工具而言，研发文化建设涉及工具、模板、规范的制定和推广落地应用，包括配置管理、持续集成、自动化交付等具体表现形式。

• 过程

对于过程而言，涉及实现方案总体设计与把控、团队实现分工分任务整体推动与协调、团



队项目复盘总结经验以及知识分享等多个方面。

- 代码

对于代码而言，代码评审、单元测试覆盖率、代码框架、前后端分析、接口定义通常需要一定的手段形成规范并执行。

- 数据

对于数据而言，包括系统缺陷率等研发质量类数据，系统接口调用性能、服务器运行状态等运行时状态数据以及日志埋点等业务类数据。

- 工程实践

对于工程实践而言，研发过程上包括协作、思考、计划、发布、开发等个维度的实践方法。典型的站立会议、迭代思想、客户测试、重构、技术预演等都属于常见的工程实践。

团队生命周期特征、产品研发特征和技术开发特征构成了研发文化的切入点，这三者之间形成一个整体。一般而言，产品研发过程驱动团队的生命周期，而产品研发特征又会对技术开发特征有较大影响。这些切入点对于微服务架构实施过程中进行研发文化的转型同样适用。

## 8.3 微服务架构转型案例分析

本节将使用一个具体的移动医疗系统案例来阐述现有系统改造的方法和实践。移动医疗系统是互联网+背景下传统医疗业务与互联网技术的碰撞，用于帮助患者解决挂号、就诊、支付、查看检查检验报告等场景中的痛点，将部分线下业务转移到线上从而提供更好的用户体验。

### 8.3.1 系统描述

作为传统行业在互联网+上的一次尝试，现有系统已经运行了一段时间，也碰到了很多问题。我们将对系统的业务领域做简单介绍，帮助读者了解系统的业务场景和功能，同时重点抛出系统中存在的各项问题，正是这些问题促使我们对系统进行微服务架构改造。

#### 1. 业务领域概述

医院门诊服务作为接纳病人的窗口，是病人了解医院、认知医院的重要入口，也是病人就诊的第一步。门诊服务窗口每天需要接待大量的就医患者，是人群高度集中而流程性较强的场所，也是医院日常管理工作的关键环节。

门诊服务管理工作繁忙、琐碎、工作重复性强而缺乏新意，同时就诊科室种类和医生等信



息多而杂，在就诊过程中就医流程冗长、现场排队严重日益成为突出的问题。在一次门诊就医过程中，挂号需要排队，缴费需要排队，取报告又要排队。一些三级医院，整个就医流程往往要三个小时以上，而真正有价值的环节，不到半个小时。一方面在患者的就医体验上，带来了非常负面的影响，使患者质疑医院的服务质量；另一方面，也需要医院大量的管理和维护投入，加重了就医过程的人力成本和流程成本。如何消除就医过程中不必要的浪费，为患者提供高效和优质的服务是医疗管理者面临的新课题，图 8-25 以门诊就医为例阐述了整个就医交互流程，其中用斜体字标注的就是存在浪费的业务场景。

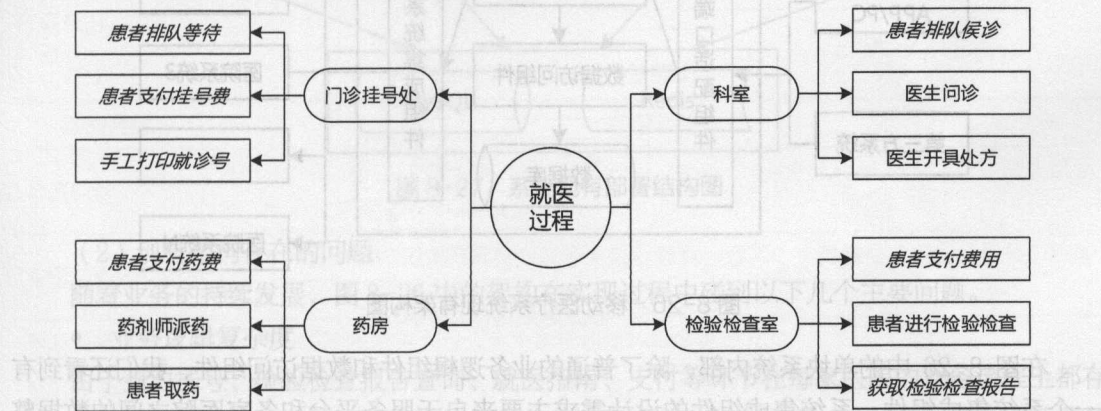


图 8-25 门诊就医交互流程

移动互联网理念和智能手机的普及为提高患者就医服务质量开辟了新模式。移动医疗系统力图使用创新的理念和技术解决传统门诊就医管理中存在的一系列问题，并创建一种高标准、高质量的门诊就医服务新模式。移动医疗系统的目标就是使用互联网技术手段，通过对接国内各个地区的医院和医疗机构，将原有分散在医院和医疗机构的医疗资源从线下走向线上，实现统一的移动医疗平台。

由于整个移动医疗系统建设涉及面非常广，本案例主要关注于向微服务架构转型这一主题，无意对所有的业务内容做过多展开。在业务上，后续内容将主要围绕预约挂号、检验检查报告查询、就医指南、支付等重点功能做分析和介绍。

## 2. 现有系统实现方案

现有架构在设计之初并没有对领域建模、系统拆分等做过多考虑，而是采用最为经典的单块系统架构，这也代表了很多需要向微服务架构转型的系统现状。在具体介绍微服务架构转型之前，有必要对现有系统做一些介绍。

### (1) 系统现有架构

图 8-26 描述了现有移动医疗系统的整体架构。我们可以看到该系统一方面需要面向客户



端系统和各种外部第三方系统提供访问接口，另一方面也需要依赖于各个医院所提供的信息化系统服务。这主要是因为移动医疗系统的业务流程很大程度上与医院存在深度耦合，需要依赖于来自于医院内部的医生、科室、号源、患者等信息，同时也需要把用户对预约挂号等的操作数据回传到医院系统。

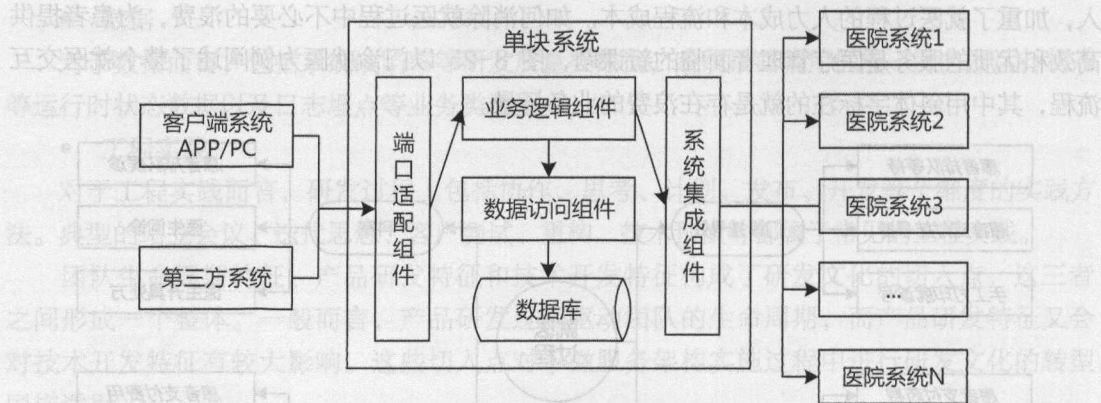


图 8-26 移动医疗系统现有架构图

在图 8-26 中的单块系统内部，除了普通的业务逻辑组件和数据访问组件，我们还看到一个系统集成组件。系统集成组件的设计需求主要来自于服务平台和各家医院之间的数据整合。考虑到各家医院内部可能采用的是不同的 HIS（Hospital Information System，医院信息系统）、LIS（Laboratory Information Management System，实验室信息管理系统）或 PACS（Picture Archiving and Communication Systems，影像归档和通信系统）系统，势必会面对各种不同的技术实现体系，系统集成的难度就体现在多个异构系统之间的数据传递和转换。

而端口适配组件面向客户端和第三方系统，采用主流的 RESTful 风格构建。表现如下，一套后台服务对接多个前端；一种数据结构，使用 JSON 作为前后端交互的数据媒介；前后端解耦，View 与 Service 高度独立。在图 8-26 中，端口适配组件依赖业务逻辑组件，业务逻辑组件依赖数据访问组件的同时也会使用系统集成组件完成完整的业务处理流程。

由于采用单块系统架构，现有移动医疗系统的部署结构相对比较简单，我们只需要对单块系统做负载均衡处理即可。图 8-27 展示了系统现有部署结构图，我们可以看到目前只采用了单个 MySQL 实例，并基于 Redis 做了一层分布式缓存。



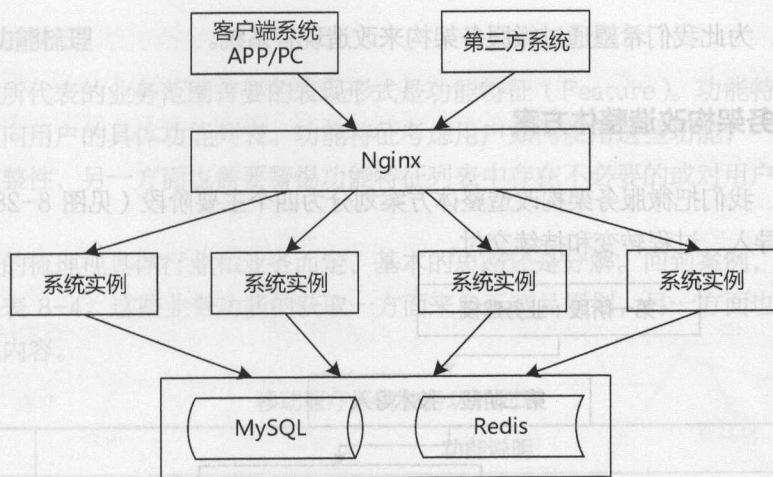


图 8-27 系统现有部署结构图

## （2）现有架构存在的问题

随着业务的持续发展，图 8-26 中的架构在实现过程中碰到以下几个主要问题。

### • 业务逻辑复杂度

由于预约挂号、检验检查报告查询、就医指南、支付等环节在每家医院的业务流程上都存在一定的差异，我们无法简单通过维护一套业务流程来应对所有场景，也就意味着系统的业务逻辑随着接入医院数量的增加也随之变得更加复杂。同时，因为在开始设计系统时无法囊括所有业务场景，后续的业务环节新增和变更会对系统实现带来很大挑战。

### • 性能

随着对接各个医院和医疗机构的数据日益增大，部分功能的响应时间已超过了正常可以接收的范围。虽然就医是一个低频事件，整体的数据量和访问量并没有那么高，但一部分服务已经成为影响系统整体性能的瓶颈所在。

### • 代码结构

由于现有代码结构是基于单块系统，所有的代码都放在一个大型工程里面，任何改动都需要对整体代码进行分支、合并等管理，过程繁琐、效率低下且容易出错。而且，系统从初始阶段运行到现在，关于模块的划分、代码的统一风格等也存在一定问题。

### • 开发团队

开发团队的变更也给系统进一步演进带来挑战。由于是一个遗留系统，很多熟悉代码整体结构的开发人员已经离职，新成员也不大倾向在原有系统上做过多抽象，除了做一些简单修改之外，转而不断开发新的实现方式，也加剧了系统结构的臃肿。

结合本章开篇提到的向微服务架构转型的原因，我们意识到现有系统已经到了需要进行重



新拆分的阶段,为此我们希望通过微服务架构来改造现有系统。

### 8.3.2 微服务架构改造整体方案

在本书中,我们把微服务架构改造整体方案划分为四个主要阶段(见图 8-28),分别为业务建模、技术导入、过程转变和持续交付。

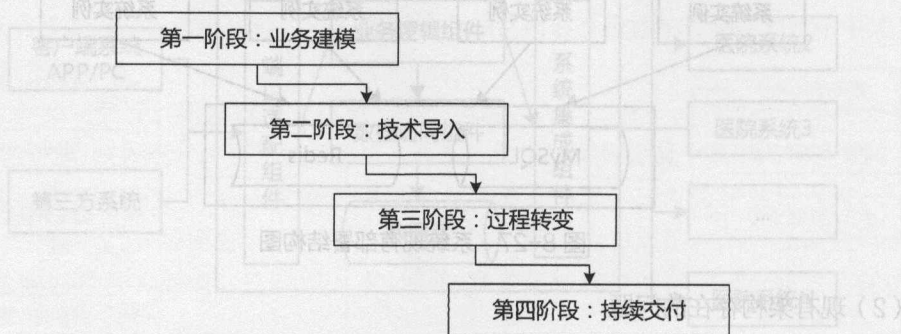


图 8-28 微服务架构改造阶段

#### • 业务建模

作为第一阶段,我们主要的切入点并不在技术而是业务。通过梳理现有业务功能、服务边界划分、服务数据规划并结合服务拆分和集成策略完成业务建模。

#### • 技术导入

完成业务建模之后,就可以进行技术导入。通过设计整体系统架构并引入主流的微服务工具和框架,我们就可以建立完善的技术体系。

#### • 过程转变

在第三阶段,我们将主导开展过程转变的相关工作。这一阶段的内容基本围绕上一节中介绍的产品管理转变、组织架构转变和研发文化转变这三个维度展开。

#### • 持续交付

在技术体系建设和过程转变完成之后,我们将关注于持续交付。在这一阶段,持续集成的交付思想和工程实践的引入以及过程资产建设等内容是工作重点。

### 8.3.3 微服务架构改造第一阶段

微服务架构改造第一阶段的主要目标是建立业务体系。微服务架构的基本思想在于考虑围绕着业务领域组件来创建应用,这些应用可独立地进行开发和管理。这一阶段的主要产出是清晰的服务边界以及各个服务之间的依赖关系。



# 1. 业务功能梳理

一个系统所代表的业务范围首要的表现形式是功能特征（Feature）。功能特征包括产品的主要特性和面向用户的具体功能列表。功能特征考虑用户如何使用这些功能，一方面需要保证功能列表的完整性，另一方面也需要警惕功能特征列表中存在不必要的或对用户而言价值较低的特征。

功能特征的梳理视具体行业和业务而定，基本的思路还是分解。回到案例，移动医疗系统的业务功能见表 8-4。这些业务功能的获取一方面来自于产品规划，另一方面也参考市面上现有系统的实现内容。

表 8-4 移动医疗系统业务功能列表	
功能名称	功能说明
普通门诊	对未来一周或当月门诊科室排班进行查询、预约、挂号并支付挂号诊疗费、获取就诊号，就诊日凭就诊号直接到科室就诊
专家门诊	对未来一周或当月门诊专家排班及号源进行查询、预约、支付挂号诊疗费、挂号、获取就诊号，就诊日凭就诊号直接到专家诊室就诊
预约挂号支付	支付普通门诊和专家门诊过程中的预约挂号诊疗费
预约挂号记录	查看预约及挂号的时间、科室、专家等信息，根据预约信息撤销或确认挂号
医院公告	查看医院面向患者的公告信息，涉及停诊、健康讲座、信息公示等信息
检查报告	查看近三个月的检查记录，及每次检查的时间、项目、检查结果等信息
检验报告	查看近三个月的检验记录，及每次检验的时间、项目、结果、参考区间等信息
费用记录	展示通过手机支付的进展情况，过往支付的记录信息查询
医院简介	查看医院简介、专长学科、医院荣誉、医院设施等概要信息
院内导航	查看院内门诊、住院、医技等楼宇的楼层信息
院外导航	提供“我的位置”到医院的最佳路线，可显示汽车、公交、步行等方式的切换
特色科室	科室列表及科室荣誉、科室骨干、擅长等介绍
名医馆	展示名医专家姓名、性别、年龄、职称、擅长等信息
健康资讯	分类发布健康、康复、养生类信息，患者可查看、评论
自助诊查	患者根据自身病情选择症状，系统依据知识库给出建议就诊的科室信息
满意度调查	患者对就医过程进行评价，医院收集并分析满意度信息
账户管理	注册账户的信息维护、密码管理、登录管理等
就诊卡管理	就诊患者及就诊卡管理，维护已有持卡人及就诊卡的信息



移动医疗系统的局部特征树。

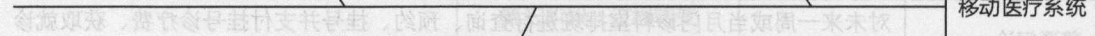


图 8-29 移动医疗系统的局部特征树

我们已经看到了一些服务与服务之间的初步划分方式和结果。

## 2. 业务边界划分

当我们面对有限的系统需求信息，其目标则是根据以下问题找到合适的答案

当我们面对有限的系统需求信息，基本思路是根据以下问题找到合适的答案。

- 核心域的通用语言是什么?

策略设计的第一步是找到系统的核心域，并根据核心域的定位和作用梳理其通用语言。核

- 核心域的支撑子域和通用子域是什么?

有了核心域，下一步就是判断系统是否只要一个核心域就能满足建模需求。如果不是，那

的服务过程！各个服务之间的依赖关系。



- 核心域如何与其他子域进行协作和集成？

系统的交互和集成以核心域为主体展开，当核心域面对支撑子域或通用子域时，使用的协作和集成策略往往是不一样的，这方面需要根据具体的子域分析和设计。

- 如何针对各个子域安排人员？

子域之间的集成不仅仅是技术问题，同样也是组织问题。围绕各个子域的划分结果，采用一个子域一个团队的模式亦或使用其他人员安排方式同样也是策略设计的期望产出。

基于以上划分子域的基本思路以及子域所代表的三种分类，我们把移动医疗系统的所有业务功能以子域的方式进行重新组织，得到如图 8-30 所示的系统子域划分方案，图 8-30 中的大部分业务功能都已经被分别归到了各个子域中。

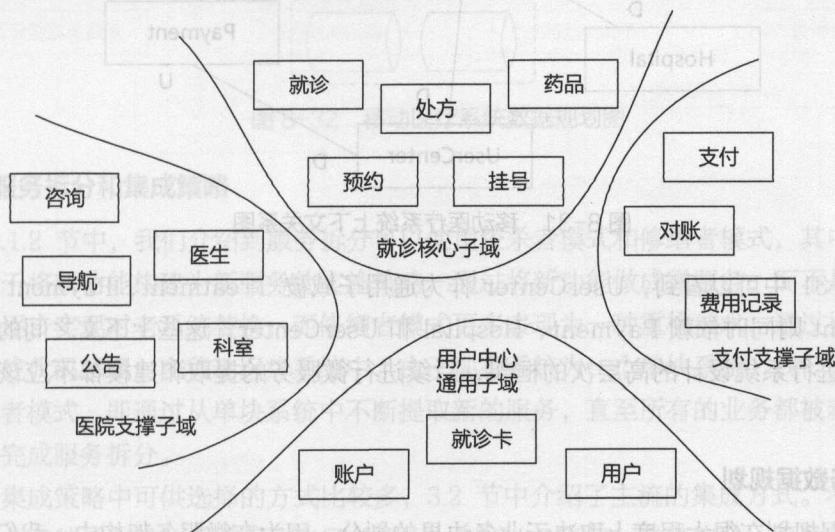


图 8-30 移动医疗系统子域拆分

- 核心子域

我们使用“就诊（Treatment）”域来命名移动医疗系统的核心子域。对于该系统而言，核心子域只有一个，因为所有其他的业务功能都是围绕患者就诊这一核心需求所展开。

- 支撑子域

关于支撑子域的划分可以比较自由，因为支撑子域为核心子域服务，但又不像通用子域那样存在较多的外部依赖性。在本系统中，我们将设计两个支撑子域，分别是“支付（Payment）”子域和“医院（Hospital）”子域。提取“支付”子域的原因在于支付业务涉及比较敏感的金钱信息，同时也会与外部第三方系统存在较多依赖，倾向于单独进行管理。而“医院”子域也存在类似的需求，但凡需要与医院进行系统集成的业务都将放在该子域中统一管理。



- 通用子域

通用子域的提取相对比较明确，涉及用户账户体系、就诊卡体系等基础信息管理类功能都可以归到这一子域中，我们将该通用子域命名为“用户中心（UserCenter）”子域，用于为其他子域提供基础服务。

明确了系统的各个子域，我们来进一步明确各个子域所代表的界限上下文，也就是明确各个子域之间的依赖关系。图 8-31 展示了移动医疗系统各个上下文之间的上下游关系。

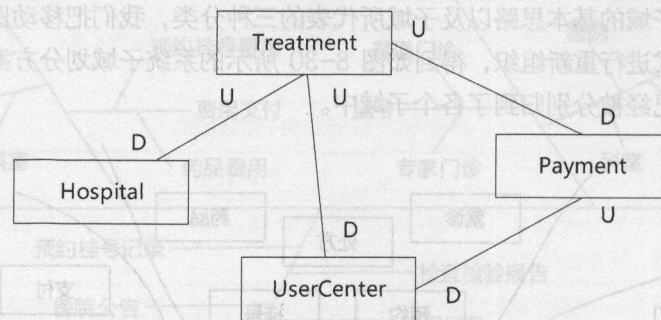


图 8-31 移动医疗系统上下文关系图

从图 8-31 中可以看到，UserCenter 作为通用子域被 Treatment、Payment 所依赖，而 Treatment 则同时依赖 Payment、Hospital 和 UserCenter。这些上下文之间的依赖关系构成了我们进行系统设计的高层次的框架，后续进行微服务的提取和建模都不应该打破这层依赖关系。

### 3. 业务数据规划

业务数据规划在很大程度上取决于业务边界的划分，因为在微服务架构中，我们认为数据和业务同属于服务的一部分。我们在 3.1.4 节中讨论了数据管理的策略，也明白数据中心化管理的优点和缺点。在这个案例中，我们将采用一种平衡思想，即基于子域管理数据，原则上在一个子域中采用单数据存储实例策略。而对于部分业务场景，如果我们认为单数据存储实例策略不能满足业务发展的需求，则可以采用多数据存储实例策略。

例如，在移动医疗系统中，核心域 Treatment 因为涉及核心业务场景，建议采用多数据存储实例策略；同样，用户中心子域 UserCenter 因为管理着整个系统的大量基础数据，通常我们也会考虑利用多数据存储实例以提高该子域的数据访问性能和存储量级。基于以上分析得出的业务数据规划方案见图 8-32。图 8-32 展示的只是站在业务层面上的一个初步数据规划结果，详细的规划我们将在第二阶段技术体系建立的过程中进一步展开。



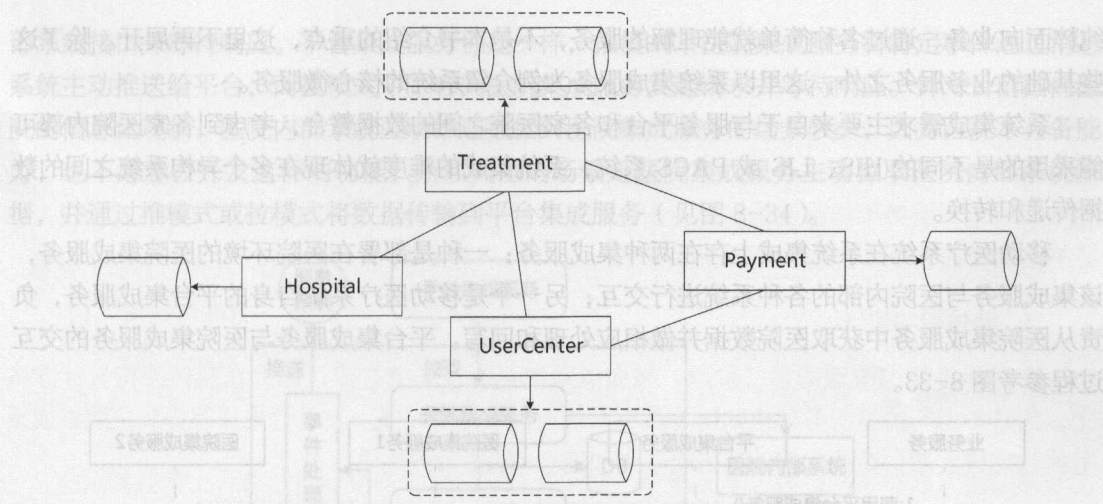


图 8-32 移动医疗系统数据规划图

4. 服务拆分和集成策略

在 3.1.2 节中，我们介绍到服务拆分可以使用绞杀者模式和修缮者模式，其中绞杀者模式的关键在于将新功能构建为新服务的实施策略，通过将新功能做成微服务，而不是直接修改原有系统，逐步实现对老系统替换；而修缮者模式更多表现为一种重构技术，通过提取抽象组件的方式完成代码解耦。在移动医疗系统中，由于现有系统为一个单块系统，我们的思路还是偏向于绞杀者模式，即通过从单块系统中不断提取新的服务，直至所有的业务都被新服务完全替换，最终完成服务拆分。

服务集成策略中可供选择的方式比较多，3.2 节中介绍了主流的集成方式。考虑到移动医疗系统的业务特性以及与医院系统之间的交互，RPC、REST、消息传递、服务总线、外部集成等都是潜在需要关注的服务集成策略。

8.3.4 微服务架构改造第二阶段

在向微服务架构转型过程中，技术的导入是相对比较容易的一个阶段。本书从第 4 章到第 6 章的内容都是围绕着“微服务实现”这一主题展开详细的介绍，包括微服务架构的基础组件、关键要素和技术框架。在微服务架构改造的第二阶段，我们的主要目标就是建立技术体系。

1. 核心微服务

建立微服务技术体系的第一步是抽取和实现系统中的核心微服务。在移动医疗系统中，我们根据图 8-30 移动医疗系统子域中的服务拆分结果可以提取很多微服务，其中绝大多数都是



纯粹面向业务、通过名称简单就能理解的服务，不是本书介绍的重点，这里不再展开。除了这些基础的业务服务之外，这里以系统集成服务为例介绍系统的核心微服务。

系统集成需求主要来自于与服务平台和各家医院之间的数据整合，考虑到各家医院内部可能采用的是不同的 HIS、LIS 或 PACS 系统，系统集成的难度就体现在多个异构系统之间的数据传递和转换。

移动医疗系统在系统集成上存在两种集成服务：一种是部署在医院环境的医院集成服务，该集成服务与医院内部的各种系统进行交互；另一个是移动医疗系统自身的平台集成服务，负责从医院集成服务中获取医院数据并做相应处理和回写。平台集成服务与医院集成服务的交互过程参考图 8-33。

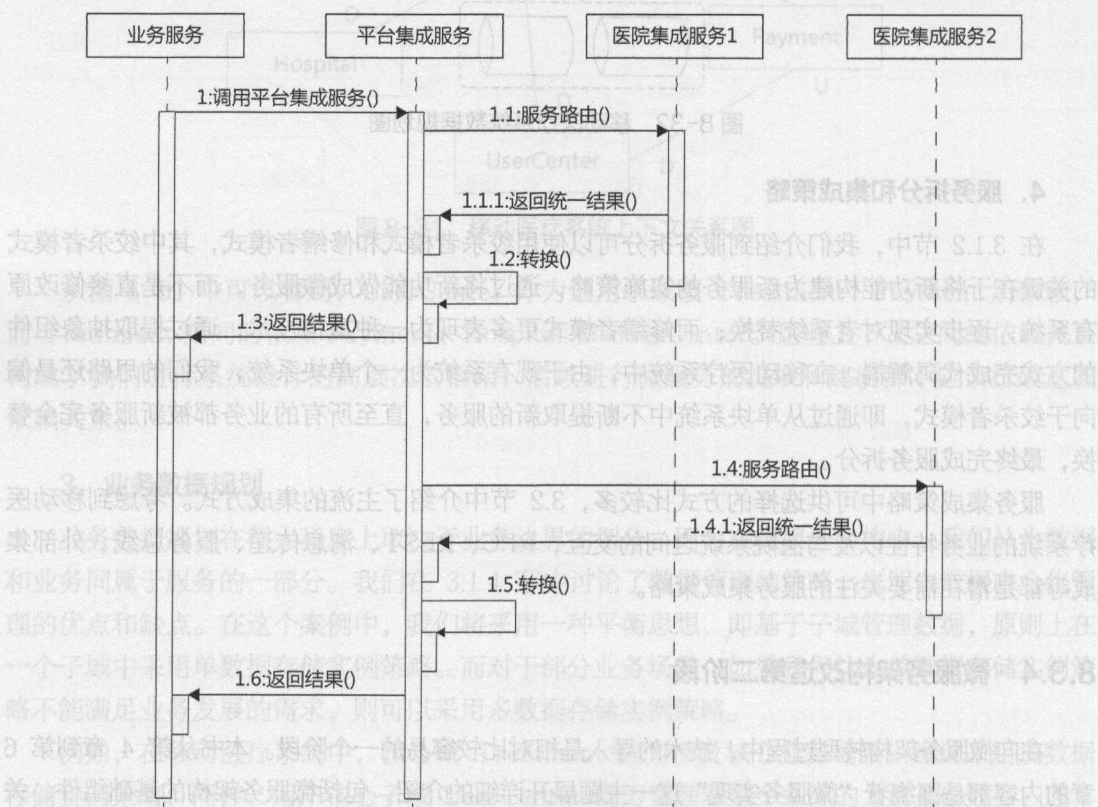


图 8-33 两种系统集成服务

针对医院集成服务和平台集成服务之间的具体交互方式，存在推（Push）模式和拉（Pull）模式两种表现形式。对于医院集成服务而言，推模式是指医院内部的 HIS、LIS、PACS 等系统主动调用医院集成服务通知事件的发生，而拉模式则是医院集成服务调用医院内



部系统接口并等待响应。平台集成服务也是一样,推模式下医院集成服务将特定事件通过消息系统主动推送给平台,而拉模式中平台向医院集成服务发起请求并等待响应。针对目前国内医院的信息化现状,医院内部系统主动推送数据到医院集成服务的场景较少,因为医院不具备能力,也不愿意去开发这样的功能,所以典型的场景是医院集成服务主动拉取医院内部系统数据,并通过推模式或拉模式将数据传输到平台集成服务(见图8-34)。

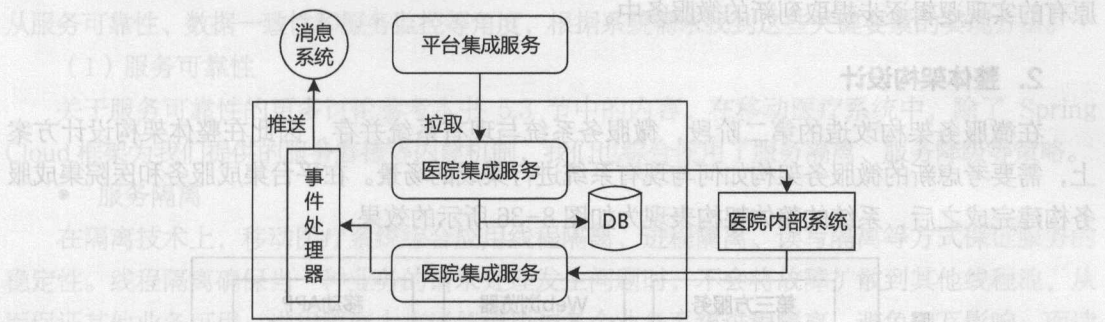


图 8-34 平台集成服务和医院集成服务交互图

对于平台集成服务而言,其本质上体现的是一种总线思想。代表各个医院系统的医院集成服务都挂载到平台集成服务这条总线上,而来自用户的具体请求将通过总线中的路由机制路由到目标医院,并进行平台集成服务与各个医院集成服务之间的数据转换,从而完成整个业务请求的链路管理。同时,平台集成服务对于上层服务而言还起到统一入口、统一格式和系统解耦的作用,所有集成操作统一由一个入口调用,所有集成操作具有一致的入参和出参,并且实现业务逻辑与集成服务之间的分离。平台集成服务的组成结构见图8-35。

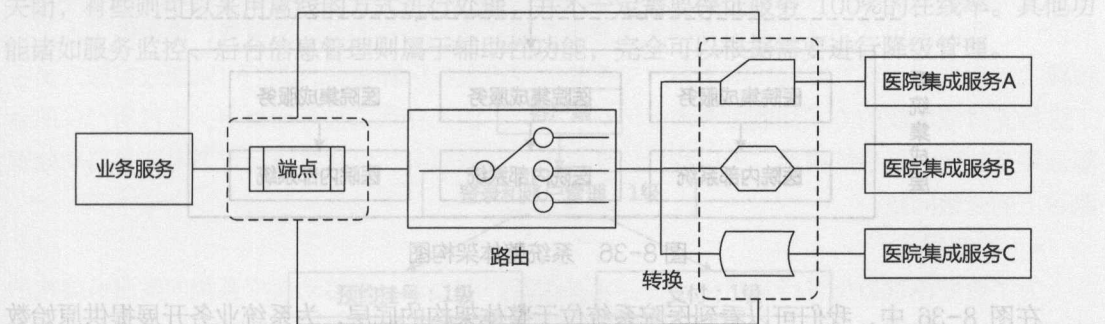


图 8-35 平台集成服务组成结构

另外,与医院的远程交互受限于每个医院网络的环境,往往是系统性能的瓶颈所在,也会直接影响用户的交互体验。为了减少远程交互的时间,网络传输的数据越小意味着传输所需的时间越少,所以为了控制传输数据的大小,我们就要选择合适的数据序列化方式。在平台集成



服务中同样需要提供医院级别的序列化支持方式,可以重点参考 protocol buffer、thrift、json、kryo 等序列化工具。我们可以根据具体医院的网络状态来进行选择,同时要求平台集成服务在设计时能够灵活支持不同的协议,这点在图 8-35 中也有所体现。

对于移动医疗系统而言,两种系统集成服务的抽取有助于更好地设计系统的整体架构,是现有系统向微服务架构改造的重要一步。其他各种业务功能也可以根据需要做相应的操作,把原有的实现逻辑逐步提取到新的微服务中。

## 2. 整体架构设计

在微服务架构改造的第二阶段,微服务系统与现有系统并存,因此在整体架构设计方案上,需要考虑新的微服务架构如何与现有系统进行集成的场景。在平台集成服务和医院集成服务构建完成之后,系统的整体架构表现为如图 8-36 所示的效果。

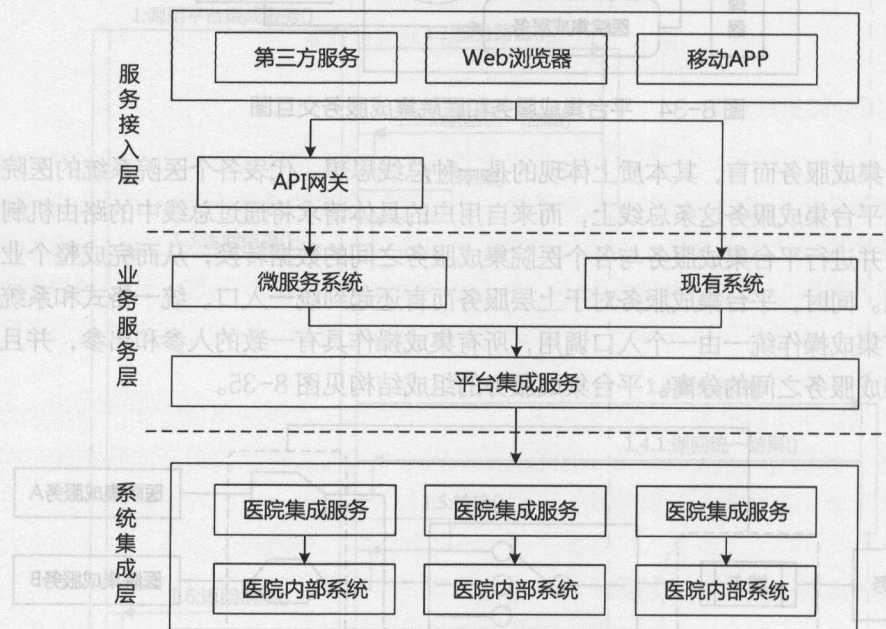


图 8-36 系统整体架构图

在图 8-36 中,我们可以看到医院系统位于整体架构的底层,为系统业务开展提供原始数据。医院系统通过医院集成服务与平台集成服务进行对接,平台集成服务在图中扮演服务总线的角色。在平台集成服务的上层,新的微服务系统与现有系统将在一段时间内并存,两者同时依赖于平台集成服务。位于整体架构图上端的是各种客户端系统,一部分通过 API 网关与微服务进行交互,而另一部分则与现有系统进行直接交互。



### 3. 技术体系建立

至此，我们已经从现有系统中分离出来了微服务，而且采用图 8-36 中的整体架构完成了微服务系统与现有系统之间的并存，后续通过持续的服务绞杀应该就可以把现有系统中的功能逐步迁移到新的微服务系统中。除了服务迁移，在微服务架构构建的第二阶段，我们还需要进一步考虑微服务架构实现上的一些关键要素，从而完成技术体系的建立。在本节中，我们将从服务可靠性、数据一致性和服务监控等角度，根据系统需求找到这些关键要素的实现方法。

#### （1）服务可靠性

关于服务可靠性的更多讨论参考本书 5.3 节中的内容。在移动医疗系统中，除了 Spring Cloud 框架为我们提供的服务容错等内建机制，我们也综合使用了服务隔离、服务降级等策略。

##### • 服务隔离

在隔离技术上，移动医疗系统综合应用线程隔离、进程隔离、读写隔离等方式保证服务的稳定性。线程隔离确保当一种业务的请求处理发生问题时，不会将故障扩散到其他线程池，从而保证其他业务可用；进程隔离上将通信模块与各个业务系统进程隔离，避免相互影响；而读写隔离则主要针对数据的读写服务隔离，我们专门设计了搜索服务用于实现读写分离。

##### • 服务分级

图 8-37 展示了在移动医疗系统中基于服务分级思想所构建的部分业务体系，我们沿用 5.3.6 节中介绍的服务分级方法把这些业务服务分成 3 个级别。我们可以参考，对于这样一个用于预约挂号的移动医疗系统而言，用户登录和账户管理、预约挂号、支付等涉及核心业务链路以及金钱方面的服务应该是优先级最高的业务功能，因为缺少这些功能，用户就肯定不会使用该款产品。而医院信息查询和历史记录分析等功能则并不属于核心业务链路，有些可以暂时关闭，有些则可以采用离线的方式进行处理，并不一定需要保证服务 100% 的在线率。其他功能诸如服务监控、后台信息管理则属于辅助性功能，完全可以根据需要进行降级管理。

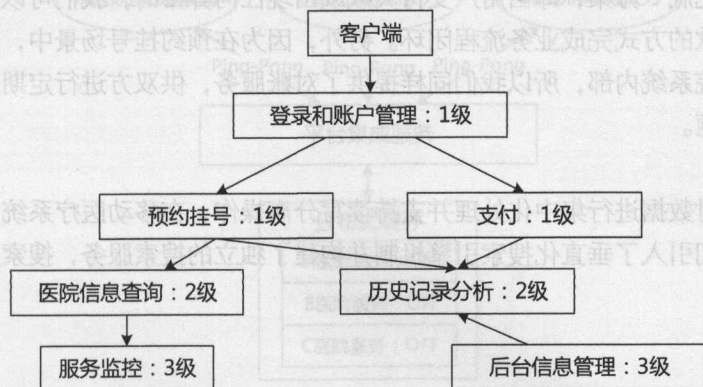


图 8-37 移动医疗系统服务分级示意图



## （2）数据一致性

移动医疗系统中数据一致性的需求来源于以下几个方面。

### • 新老系统数据耦合

移动医疗系统中，新的微服务系统与现有系统之间存在一定的数据耦合，需要考虑两个系统之间的数据同步问题。

### • 分布式服务数据一致性

用户在预约挂号的同时需要进行支付操作，根据我们在第一阶段中的服务边界划分结果，预约挂号服务属于“就诊”子域，而支付服务属于“支付”子域，这就构成了典型的分布式服务之间的数据一致性问题。

### • 统一化数据查询需求

对大多数业务系统而言，查询是一种高频操作且要求具备很高的灵活性，通常需要进行数据聚合。对于预约挂号场景而言，需求可能更为复杂，因为部分业务数据是通过平台集成服务从医院系统中获取，我们需要对这些来自外部的数据和系统自身的业务数据进行处理并提供统一化查询入口。

针对这些数据一致性需求，我们在向微服务架构改造的过程中，采用了如下实现策略。

### • 数据复制

在本章 8.1.2 节中，我们介绍了数据分离的实现方式，并通过数据复制完成不同系统之间的数据同步。可以认为数据复制是共享数据库模式的一种改进，比较适合于新老系统之间存在的数据耦合场景。

### • 支付服务

在用户通过预约挂号入口提交订单并支付场景中，我们重点对支付服务做了数据一致性上的设计。我们采用了 5.2.3 节中介绍的可靠事件模式完成事件的发送和消费，也提供了人工干预模式以实现“兜底”方案，即当用户支付失败或出现任何异常时，我们可以通过客服人员手工进行支付或退款的方式完成业务流程闭环。另外，因为在预约挂号场景中，支付所产生的金额最终是转到医院系统内部，所以我们同样提供了对账服务，供双方进行定期对账并自动处理账目上的金额问题。

### • 搜索服务

为了更好地对数据进行集中化处理并支持读写分离操作，在移动医疗系统向微服务架构的转变过程中，我们引入了垂直化搜索引擎机制并构建了独立的搜索服务，搜索服务的基本结构见图 8-38。

图 8-38 展示了微服务架构中搜索服务的结构。在平台集成服务层，新的微服务系统与传统系统并存，两者同时依赖于平台集成服务。上层的是各种客户端，通过 API 网关与微服务进行交互。图 8-38 图



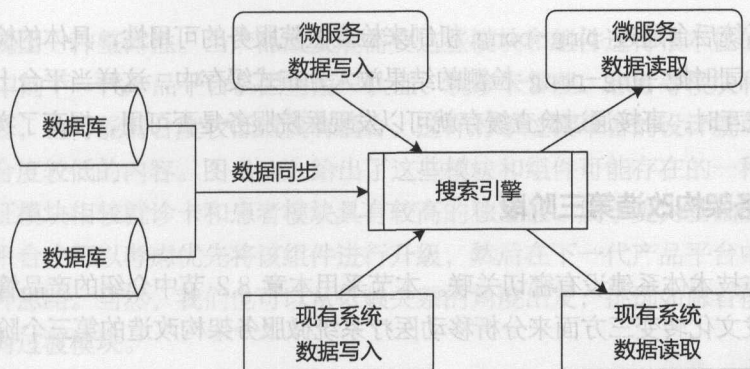


图 8-38 搜索服务结构图

在图 8-38 中，一方面，微服务系统可以往搜索引擎中添加数据和搜索数据，一般会有不同的服务分别负责数据的写入和数据的读取，从而实现读写分离。这点对于现有系统同样适用。另一方面，我们也看到不同数据库中的数据也可以通过数据同步的方式导入到搜索引擎中，从而供微服务或现有系统使用。

### （3）服务监控

在移动医疗系统中，有一个服务需要从业务层面进行监控，这就是医院集成服务。由于网络原因或者医院本身提供的对外服务接口的不可用性，将会导致某些需要接入医院网络的服务不可用。当这样的情况发生时，为了提高用户体验，同时避免不必要的麻烦（如付费成功以后才发现医院的挂号服务不可用而引发的后续一系列繁琐的步骤），移动医疗系统需要及时发现哪些医院的服务不可用，同时能明白地告知用户发生了什么。为此需要提供针对医院服务级别的监控机制。实现服务监控最常见的方法就是心跳检测，图 8-39 展示了一种心跳检测的示意图。

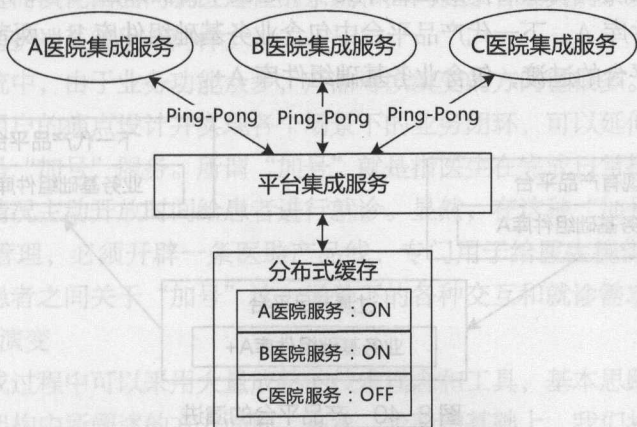


图 8-39 系统集成服务心跳检测



移动医疗系统后台通过 ping-pong 机制来检查医院服务的可用性，具体的检查时间可根据具体情况设置。同时把 ping-pong 检测的结果放入分布式缓存中，这样当平台上提供的某些服务需要和医院交互时，直接通过检查缓存就可以发现医院服务是否可用，提高了实时响应性。

### 8.3.5 微服务架构改造第三阶段

过程转变与技术体系建设有密切关联。本节采用本章 8.2 节中介绍的产品管理转变、组织架构转变和研发文化转变三方面来分析移动医疗系统微服务架构改造的第三个阶段，也就是过程转变的阶段。

#### 1. 产品管理演变

我们将基于 8.2.1 节中提到的产品化框架介绍产品管理演变的方法。

##### (1) 产品平台演变

产品平台包括的内容一方面可以是子系统、模块、组件，这些要素都完全面向业务，通常不能独立提供服务，而需要基于组合理念进行整合应用。在移动医疗系统中，与用户认证、就诊卡管理、病人等内容相关的模块和组件可以放到产品平台中，这些模块和组件自身并不能独立运行，但在预约挂号、检查检验报告查询过程中都需要基于它们才能形成一个完整的业务闭环。

另一方面，产品平台还可以上升到更高的地位，因为产品线和具体某个产品本身在不断演进，其所依赖的产品平台也需要随之演进，这种演进往往并不仅是个别或一批模块和组件的演进，而是将它们组合起来形成一个具有统一版本的整体概念。我们需要将现有平台进行演进以形成下一代平台，下一代平台一般会采用一个完全不同的结构来取代原有平台，为防止在下一代平台和现有平台之间出现空档，往往会开发一个过渡平台。如图 8-40 所示，现有产品平台中包含业务基础组件库 A，下一代产品平台中包含业务基础组件库 B，两者之间的过渡产品平台则是对现有产品平台的过渡，包含业务基础组件库 A+。

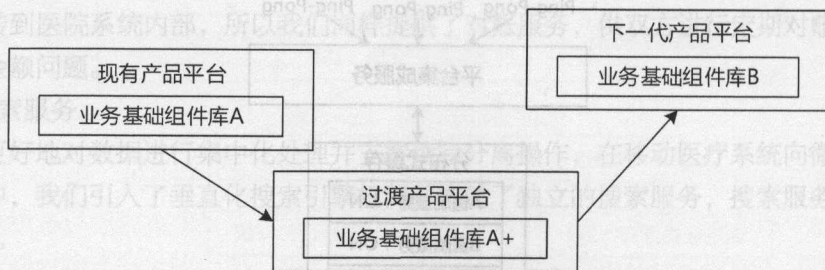


图 8-40 产品平台的演进

我们再来看移动医疗系统，用户认证、就诊卡管理、患者等模块和组件在各自独立演进的



过程中,还表现出一种整体性。当产品线发展需要这些模块和组件进行根本性的重构时,就会形成图 8-41 中的下一代产品平台。因为用户认证、就诊卡管理、患者等模块和组件之间可能存在较强的耦合,也可能耦合度较低或没有耦合,这样对于过渡平台的设计就可以先替换那些与其他模块耦合度较低的内容。图 8-41 给出了这些模块和组件可能存在的一种产品平台演进过程,用户认证模块相较就诊卡和患者模块具有较高的独立性,属于更为基础的业务模块,所以在过渡产品平台中可以考虑优先将该组件进行升级,然后在下一代产品平台中替换所有业务组件,这是一种思路。当然,我们也可以从依赖关系的角度出发,挑选如患者模块这种较少被依赖的模块作为过渡模块。

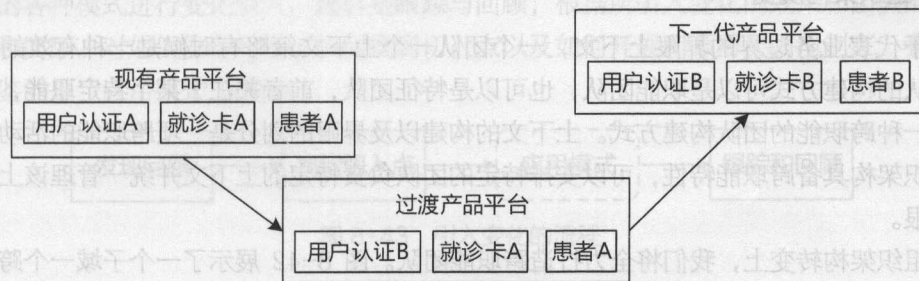


图 8-41 移动医疗系统产品平台演进示例

## (2) 产品线演变

产品线的设计和实现因产品而异,很难给出一个统一的标准。以医疗行业而言,移动医疗相关的患者端和医护端 APP 可以作为两个独立的产品并形成一条产品线,因为这两个 APP 背后的基础数据可能是同一份。但同时,我们也可以根据需要把患者端和医护端 APP 拆分成两条产品线,患者端进而演化出面向就医过程的系统和面向健康管理类的系统,从而形成一条独立发展的产品线,医护端系统也是同样。

在移动医疗系统中,由于业务功能众多,产品可以演变的方向也很多。对于预约挂号等业务场景,如何围绕用户的痛点设计并实现各个场景下的业务闭环,可以延伸出很多业务流程,比较有代表性的就是“加号”服务。所谓“加号”就是指医生在完成日常挂号量的前提下,可以根据自身的工作情况主动开放时间给患者进行就诊。显然,在这种“加号”场景下,我们想要实现业务的闭环管理,必须开辟一条医助产品线,专门用于给医生提供能够“加号”的入口,并管理医生与患者之间关于“加号”这一场景下的各种交互和就诊需求。

## (3) 技术平台演变

技术平台在构成过程中可以采用大量成熟的技术理念和工具,基本思路就是实现服务化,也就是本书微服务架构中所阐述的方方面面。在第二阶段的基础上,我们将把技术平台建设作为一项独立的工程进行管理和推进。



在移动医疗系统中，部分技术体系也可以上升到技术平台的高度进行统一管理。比较典型的就是系统集成组件，我们在第二阶段中已经推进了系统集成组件的改造并抽取了平台集成服务和医院集成服务等多个微服务，其中平台集成服务比较适合用于建设团队自身的技术平台。一方面，平台集成服务属于基础技术型服务，并不仅仅适合实现移动医疗系统中的某一个产品或产品线，而是可以用作对总线相关需求的高度抽象。另一方面，平台集成服务中的路由、转换等功能具有高度通用性和扩展性，但又很难在短时间内完美实现所有功能组件，一般都需要在系统演进过程中进行不断扩充和优化，需要作为技术平台对其进行独立规划。

## 2. 组织架构演变

关于代表业务边界的界限上下文，一个团队一个上下文策略有时候是一种有效的拆分策略。团队的构建方式可以是职能团队，也可以是特征团队，前者关注于某个特定职能，而后者则代表一种跨职能的团队构建方式。上下文的构建以及界限的划分是一项跨职能的活动，如果团队组织架构具备跨职能特性，可以安排特定的团队负责特定的上下文并统一管理该上下文对应的界限。

在组织架构转变上，我们将全力打造跨职能团队。图 8-42 展示了一个子域一个跨职能团队的构建方式。

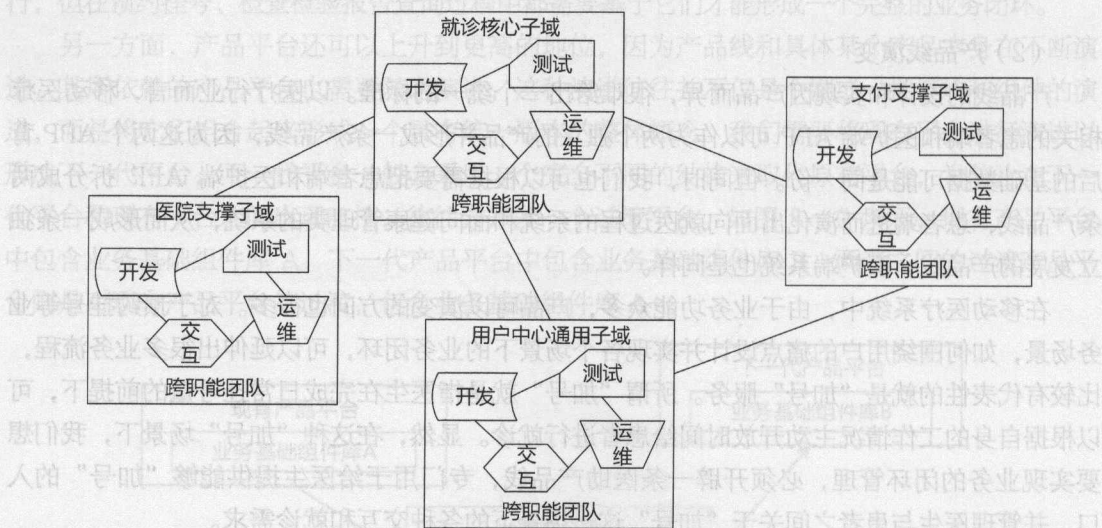


图 8-42 跨职能团队与子域

康威定律告诉我们，你想要什么样的系统设计，就组建什么样的团队，能扁平化就扁平化。按业务划分团队，这样能让团队自然地自治内聚，明确的业务边界会减少和外部的沟通成本，每个小团队都对自身模块的整个生命周期负责，没有边界不清的问题，也就避免了很多团



队之间无谓的扯皮现象。

### 3. 研发文化演变

向微服务架构转型的过程是一个不断尝试的过程，也可以理解为是一种不断引入变化的过程，势必需要导入很多新的理念、思想和工程实践，从而对现有的研发文化造成冲击。引入变化是为了过程改进，其基本的流程如图 8-43 所示，每个步骤都需要专职角色进行负责和把控。其中发现问题是指针对团队中的“Bad Smell”，提取为需要引入变化进行解决和改善的问题点，然后通过收集、分析数据对问题点进行剖析并找到切入点。应用模式上，组合应用下文提到的各种模式进行变化引入。最后是跟踪与回顾，根据所引入变化的效果和团队的反馈进行总结和回顾。对如何发现问题、如何找到切入点以及如何开展回顾需要因地制宜，我们可以直接借鉴的是“应用模式”部分。



图 8-43 引入变化的流程

在一个特定的公司文化中，真正可以推动研发文化建设的工作主要体现在精神层面和制度层面两个方面。在本节中，我们将从引入变化的角度出发讨论如何在精神层面和制度层面加强研发文化建设。

#### （1）精神层面

充分认识到开发工作的重要性是至关重要的，但并不是每个技术管理者都会意识到这一点。国内的大多数企业，传统行业偏向销售市场驱动，而互联网行业则由运营和产品来推动整个研发工作。技术团队普遍处于整个研发流程的末端，地位偏低没有话语权。这个时候，技术管理者需要能够站在开发人员的立场看问题，竭力支持团队开发工作的开展。

在精神层面塑造研发团队文化最好的方法是技术管理者的率先垂范和团结一致，而不是期待开发人员的自觉性。其次，由于开发活动的特性，要鼓励团队成员的创新活动。鼓励团队成员的良好沟通和共同协作，即团队精神。要求团队重视工作细节和不同的意见；要强调团队成员强烈的时间观念和责任意识。在精神层面上，我们可以通过引入一定的变化来改善当前的研发文化。这方面引入变化的模式包括以下几种。

#### • 关系网

关系网模式和“自下而上，全员参与”这一思路有关，有时候变化想要获得大家的认可，需要有人来帮你造势和宣传。如果一个人在团队和组织中人员人缘很好，而且对微服务架构有清晰认识，那就建议由这个人去推动一项新想法。当然，作为团队的管理者自然会有很多的机会接触到其他团队在实施微服务架构上的做法和成果，把别的团队的新想法吸收到自己团队中



或者把自己团队中好的做法推广到其他团队也是管理的一项内容，这个时候合理利用关系网这一模式会起到潜移默化的作用。

- 学习小组

很多组织会设立类似的“学习小组”，有些模型也主张要由一个小组来统领团队的过程改进工作，如 CMMI 的 SEPG。我们这里的“学习小组”不强调类似正式的组织性架构，而是指一帮有共同兴趣的团队成員可以组成一个小组，让他们能够有机会持续地对微服务架构做一些学习和交流。当然我们认为管理工作不能少，所以定期/不定期的组织一下小组会议还是重要的，最好这个组织会议的人不是来自学习小组中的成员。

- 个人沟通

如果你想让别人听你的，请你保持与别人的沟通，因为要想说服一个人接受新想法，一定要让他了解这个新想法对他个人的好处。如果你只是在团队培训时提到了这个新想法，大家都觉得它不错，然后你就指望所有人都能去实现这个新想法，这是不现实的。尤其是研发团队中往往存在不善交际的成员，他们在开会等群体活动中并不一定会把自己的想法说出来，但不代表他们没有想法，如果不去找他们沟通，他们就不会把这些话说出来。研发团队中有时出现的各种问题就是因为很多人没有把该说的话说出来所导致，这时候专职负责人必须要灵活使用“个人沟通”这一模式确保团队中所有人的想法都公开和透明，这点在引入微服务架构的初始阶段非常重要。

## （2）制度层面

制度层面的重点在于建立适合的产品开发管理机制和研发团队绩效考核机制，并且依据考核结果做到奖罚分明。建设开发团队的工作机制，其目的在于沟通信息、明确责任、协调进度。工作机制可以分为两种，即正式机制和非正式机制。正式机制多体现为组织级别的规章制度，非正式机制则是不同部门的开发人员之间的随机交流。对于那些非正式的团队机制是不好用制度规定下来的，非正式的团队机制在很大程度上受到团队文化的制约。

在制度层面，塑造团队研发文化的主要方式就是建立各种适合自身团队发展的非正式团队机制。关于如何建立这些非正式团队机制也存在一些引入变化的模式，包括以下几种。

- 团队培训

这点相信大家没有异议，这一模式一般用在新想法的快速推广。如何快速、有效地把变革的思想和做法落实到团队，让团队中所有人都能达到统一认识水平，团队培训必不可少。通常由专职负责人起草新想法的推广方案，通过评审之后即可开展团队培训，方式也不局限于传统的一对多培训方式，可以有发散和变通，如头脑风暴、焦点小组等，都能起到不错的效果。微服务架构转型的各个方面都可以通过团队培训加强团队成员的统一认识。

- 专职负责人

从流程执行、阶段评审、高效决策等角度出发，我们都深信有或没有专职负责人是不一样



的，引入变化也是一样。对引入变化而言，对该变化持支持态度并有强烈兴趣的人无疑是专职负责人的候选。从这个角度讲，不同的问题和切入点、不同模式的专职负责人都可能不一样，所以专职负责人通常是流动和不固定的，鼓励团队中的不同成员成为这个专职负责人也是一项最佳实践。专职负责人的作用和职责就是确保在团队内部引进新想法过程的有效性，努力把新想法纳入正常工作并对该想法的落实、跟踪和调整有明确的思路。

#### • 回顾时间

有时候新想法已经引入但效果不一定理想，需要我们持续关注工作的进展并对实施过程中碰到的问题进行总结时，可以引入“回顾时间”模式。我们通常通过回顾会议的方式开展回顾活动。回顾会议的流程可以分为以下 5 个步骤<sup>[29]</sup>，其中前两步需要在会议前完成：确定目标，确定本次回顾会议中需要改进的目标，一般一次回顾 1~2 个目标比较合理；收集数据，为了对改进过程有量化的标准，需要进行数据收集，数据来自日常研发过程中与该改进目标相关的方方面面；激发灵感，使用数据进行集思广益，通过各种激发灵感的工具和活动让团队成员提出自己的想法；决定做什么，收集团队中的灵感并进行讨论和分析，最终确定下一步的目标；总结收尾，总结本次回顾会议的过程和成果并落实行动计划和责任人。回顾会议的目的是对研发过程中的客观数据进行分析从而得出结论和行动计划，当然各个团队可以根据实际情况进行调整和裁剪，但会议的输入、输出和议程应保持一致。在向微服务架构转型过程中，建议定期开展回顾会议并维护一份回顾列表，以便对回顾事项进行确认和跟踪。

### 8.3.6 微服务架构改造第四阶段

在微服务改造的第四阶段，我们重点关注的是服务的持续交付。同时，作为整个改造过程的最后一个阶段，我们也需要站在技术管理的角度出发，对转型过程中的各项过程资产进行梳理和建设。

#### 1. 持续交付

持续交付（Continus Delivery，CD）是一种软件开发实践，它所描述的软件开发是从原始需求识别到最终产品部署至生产环境这个过程中，需求以小批量形式在团队的各个角色间顺畅流动，能够帮助我们解决服务发布和交付中的一系列问题。

持续交付的一个基本思路就是要做到自动化发布。自动化代表可重复，并注重过程，过程对则结果一定对。在自动化发布环境下，无论什么修改都应该触发相应的监控流程，确保问题在第一时间被暴露和解决。

持续交付的另一个基本思路是频繁发布。一旦建立起自动化发布平台，就可以通过频繁发布降低出错所引起的风险，而频繁发布能够促进快速反馈，从而推动过程改进。



为了实现自动化发布和频繁发布，软件交付存在一些共性原则，包括为软件发布创建可重复的过程、把所有可能改变的东西都纳入版本控制、提前并频繁的进行发布演练。同时，交付过程被认为是开发、测试、运维、产品等全员的责任。DevOps（Development+Operations）就是这种思想的体现，DevOps 是一组过程、方法与系统的统称，用于促进开发、技术运营和质量保障部门之间的沟通、协作与整合。结合目前主流的敏捷思想，我们可以把 DevOps 与其他方法论以及各个角色之间的关系描绘成图 8-44。

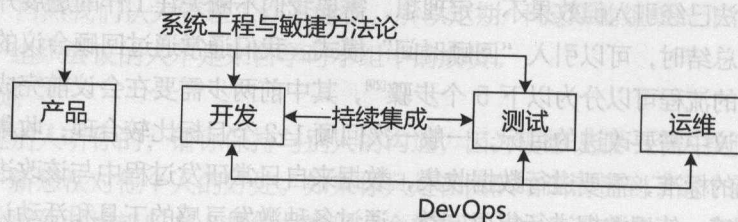


图 8-44 持续交付角色与关系

持续交付虽然是一个独立主题，但本书关于服务建模、服务拆分和集成以及采用的具体开发框架实际上都与持续交付有一定的关联。基于持续交付理念，我们根据软件开发的阶段来梳理一下在微服务开发过程中如何有效实现持续交付。

### （1）开发与测试

在开发阶段，采用软件配置管理相关工程实践能够为后续的交付工作提供良好的基础。在本书 7.2.1 节中对配置管理做了简要介绍。配置管理存在一系列的模式与最佳实践，例如，我们在使用 Spring Cloud 等框架进行微服务开发时，如何使用 Git 等版本控制系统工具规范主干（master）代码和各个开发分支（branch）之间的关系、如何采用个人工作区间与 Git 服务器之间的代码同步策略等都属于这一范畴。

在开发阶段，把握如何描述微服务、如何管理微服务的配置信息、如何对服务进行测试等也有助于更好地实现持续交付。关于这些内容，我们在本书的其他章节中都做了详细介绍，这里不再展开。

### （2）构建与集成

关于持续集成我们也已经在 7.2.1 节中进行了介绍。在持续集成实现上，相关的工具有很多，可供选择的不下数十种，如 Redmine、Maven、SVN、Git 等都很常用，而对于专用的持续集成服务器，目前最流行的当属 Jenkins。

Jenkins 是基于 Java 开发的一种持续集成工具，包括持续的软件版本发布和监控外部调用的执行情况。Jenkins 本身并没有自动构建的功能，而是将 JDK、Maven 等工具集成起来，将持续集成的整个过程可视化。

Jenkins 包括构建任务、自动化测试、通知、代码质量和自动部署与发布等方面的功能，



配置是使用这些功能的基本方法。Jenkins 提供了全局配置功能,支持系统环境相关的全局变量和 JDK、Maven 等构建工具和 SVN 等版本控制工具的全局设置。

参数化构建是 Jenkins 支持动态构建的表现形式,即通过设置外部参数影响构建过程。对于构建过程的工作流,Jenkins 也支持构建步骤和构建后操作的设置以提高构建灵活性,代码打包、JUnit 支持、发送邮件、诸如 Checkstyle、PMD 和 FindBugs 等代码质量工具的嵌入都属于构建后操作的典型应用。Jenkins 也提供了部署插件支持面向 Maven 私服和应用服务器的自动化部署。

### (3) 部署与运维

在微服务架构中,因为服务数量可能很多,相较单块系统提出了更高的要求,环境的安装、配置、服务部署等方面成本也越来越高。所以基础设施的自动化变成很多团队的一项重要工作,很多团队都会尝试使用 Chef、Puppet、Ansible 等工具完成软件的安装、配置以及部署的自动化。

如果使用 Docker 容器技术,服务构建出来的结果是一个镜像文件,该镜像文件能够运行在任何装有 Docker 的环境中,有效解决以往因为配置管理不到位所带来的开发和部署环境不一致的问题。同时,Docker 技术与虚拟化技术相互配合,也能有效提高服务器的利用效率。

对于开发人员而言,微服务的运维工作主要有两个方面:一方面在于监控服务当前的运行状态,确保能够接收和处理各种报警信息;另一方面在于高效获取服务运行的日志信息,便于可视化的分析和处理异常日志,目前主流的 ELK、Apache Flume 等工具和架构能够帮助我们实现日志聚合。在微服务架构中,服务监控是一个非常重要的话题,我们已经在 7.3 节中做了相应介绍。

## 2. 过程资产建设

在上一阶段我们讲到向微服务架构转型的过程实际上是一个引入变化的过程。很多变化引入的失败并不在起点,而是在过程的持续性上。

### (1) 过程资产的理念

组织过程资产(Organizational Process Assets)是项目管理体系的一个核心概念,在向微服务架构转型的过程中,同样需要从技术角度出发进行过程资产建设,通过梳理、设计组织过程资产在实际研发过程中的分类标准以及各类组织过程资产的来源渠道,使得开发团队对抽象的组织过程资产概念有了相对清晰、具象的认识,能够在一定程度上提升组织中关于组织过程资产沉淀的理解及执行力度,为团队的持续学习和发展奠定基础。

所谓组织过程资产是指一个学习型组织在项目和产品开发过程中所积累的无形资产。其具体的表现形式可以有两大类<sup>[27]</sup>,一类是组织指导工作的过程和程序,另一类是存储和检索信息



的组织公用知识库。前者关注过程，后者关注数据。

在从现有系统向微服务架构转型过程中，如果研发团队中出现以下症状或表象就说明研发过程资产建设是欠缺的：

- 没有文档记录导致过程资产丢失
- 开发人员个人素质要求偏高
- 交接困难，新员工难以熟悉服务的设计思路
- 缺少设计评审，设计问题延后暴露

过程资产建设是一个组织级别的活动，对研发管理而言重点关注以下几点。

#### • 知识服务于业务

从具体业务出发管理研发知识，而不是从技术出发。技术人员在进行知识梳理的过程中（尤其是刚开始的阶段），往往习惯于从技术本身出发来看问题，导致梳理出来的东西只有技术团队内部有限的几个人能看懂。业务领域模型才是一个系统的核心，技术只是这一模型的一种实现方式，技术人员梳理的知识同样也要让产品线、项目线能够参与讨论和总结。

#### • 系统运作与界限

关注系统的运作流程以及服务提供的界限。对多团队协作的研发过程而言，系统集成是团队之间协作的根本任务，如何定义系统之间的服务边界，并把这些边界梳理成统一接口进行维护是团队作为服务提供者对外所需要暴露的知识；对内而言，各个服务的逻辑流程是确保团队内部高效运作的基础。

#### • 通用库

通用库泛指通用功能、模板和流程，其包含的内容可以有多种，代表性的有过程资产定义的格式和团队交流采用的信息传递模板（如 Word、Excel 等文档的样式和风格、基本章节的划分和定义）、工作流程（如评审会议的召开方式、频率）以及在团队和组织级别进行提取的通用功能（如各个系统都能采用和适配的账号功能）。

### （2）过程资产建设工程实践

工程实践是代表知识管理思想的具体产物，对于微服务架构的实施和推行，我们可以采用的工程实践包括以下几点。

#### • 服务责任制

服务责任制的初衷是确保知识管理过程的完整性，无论领域模型、业务流程还是其他多个知识维度都通过服务这一基本单元进行组织。服务责任人负责整个服务的知识管理，在具体文档的组织上也建议一个服务维护一份文档。服务责任制的示意图参考图 8-45。



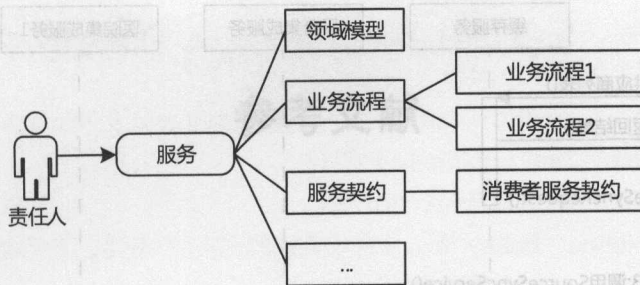


图 8-45 服务责任制示意图

### • 领域模型

按照领域驱动的设计思想<sup>[13]</sup>，一个领域模型中应该包括上下文、实体、事件、存储库等内容，其中最常用也最必要的就是上下文领域模型。图 8-46 是一个典型的上下文领域模型示例图，描述了移动医疗系统中的用户中心子域。

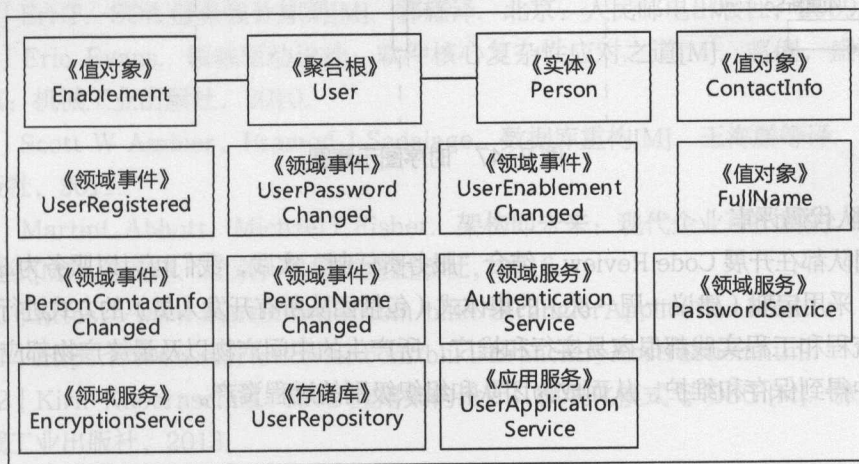


图 8-46 上下文领域模型示意图

### • UML 建模

统一建模语言是用来对软件系统进行可视化建模的一种语言，常用的用例图、时序图、状态图和活动图等在一定程度上同时面向业务和技术，其中时序图主要展示业务间的详细流程以及流程中不同对象间的调用关系和调用顺序。结合把控上文中“系统运行和边界”的知识管理思想，时序图能够详细描述前端调用接口，定义接口名称、调用方式及输入输出条件；同时也能描述后端各组件之间的调用关系及调用顺序，直观展现了业务模型，是 UML 中最适合做研发知识管理的一个必备工具。图 8-47 是用 Astah UML 工具制作的用于医院号源同步场景的时序图示意图。



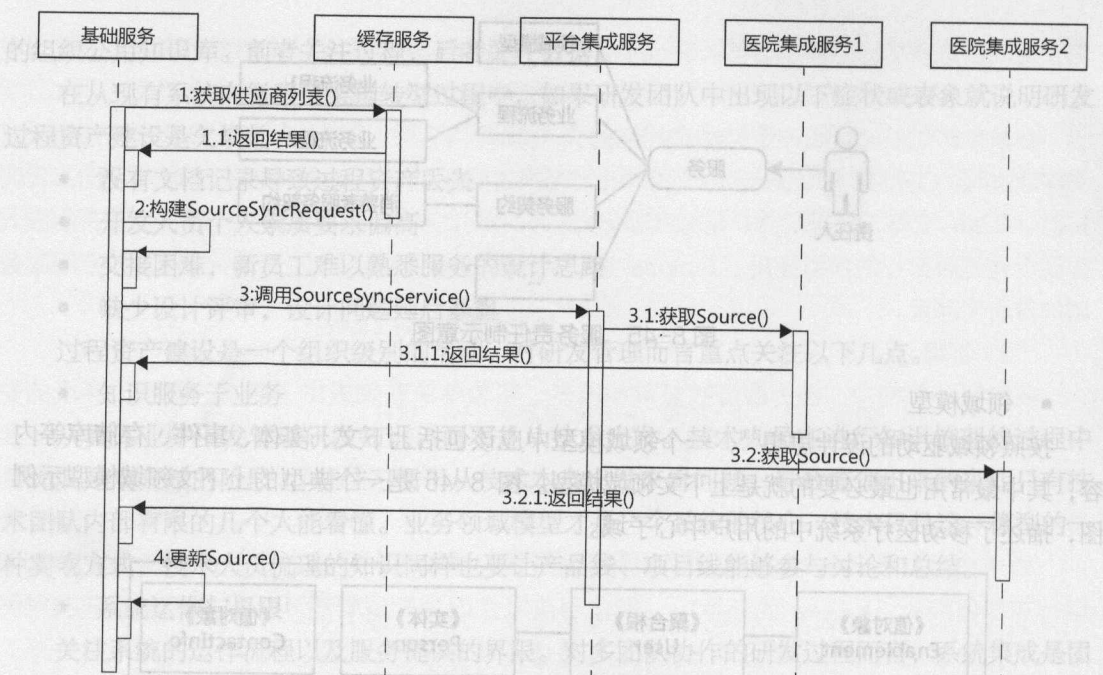


图 8-47 时序图示意图

#### • 团队代码评审

很多团队都在开展 Code Review，结合“服务责任制”实践，我们可以以服务为 Review 的基本单元，采用定期（建议一周一次）的集体式（包括团队所有开发人员）的方式进行评审。

以上流程和工程实践都很容易实行和推广，所产生的中间产物以及最终产物都应该在知识管理平台中得到保存和维护，从而成为团队和组织级别的过程资产。

## 8.4 本章小结

作为全书的最后一章，本章整合全书其余各章关于微服务架构的介绍内容，给出现有系统如何向微服务架构转型的过程和方法。

首先，我们可以通过应用一些调整架构的技术实现现有系统与微服务架构之间的并存和过渡，这是技术维度的转变。但是对于实施微服务架构而言，我们还需要关注研发过程的转变，这其中就包括产品管理的转变、组织架构的转变以及研发文化的转变。

在本章中，我们也给出了微服务架构转型的通用四阶段模型，包括业务建模、技术导入、过程转变和持续交付。针对不同阶段，我们通过一个完整的案例介绍了具体的实施要点和策略。



非卖品！！严禁（售卖和上传互联网平台）！！  
仅供对书籍质量进行鉴定甄别！为是否购买正版实体书提供依据！！

## 参考文献

- [1] <https://www.martinfowler.com/articles/microservices.html>.
- [2] 郑天民. 系统架构设计：程序员向架构师转型之路[M]. 北京：人民邮电出版社，2017.
- [3] Sam Newman. 微服务设计[M]. 崔力强，张骏译. 北京：人民邮电出版社，2016.
- [4] [http://www.melconway.com/Home/Conways\\_Law.html](http://www.melconway.com/Home/Conways_Law.html).
- [5] <https://martinfowler.com/bliki/MicroservicePremium.html>.
- [6] Erl T. SOA 服务设计原则[M]. 郭耀译. 北京：人民邮电出版社，2009.
- [7] Eric Evans. 领域驱动设计：软件核心复杂性应对之道[M]. 赵俐，盛海艳，刘霞译. 北京：机械工业出版社，2010.
- [8] Scott W Ambler, Pramod J Sadalage. 数据库重构[M]. 王海鹏等译. 北京：机械工业出版社，2011.
- [9] MartinL.Abbott, MichaelT.Fisher. 架构即未来：现代企业可扩展的 Web 架构、流程和组织[M]. 陈斌译. 北京：机械工业出版社，2016.
- [10] <https://www.martinfowler.com/bliki/StranglerApplication.html>.
- [11] <https://www.martinfowler.com/bliki/BranchByAbstraction.html>.
- [12] Kirk Knoernschild. Java 应用架构设计：模块化模式与 OSGi[M]. 张卫滨译. 北京：机械工业出版社，2013.
- [13] Vaughn Vernon, Linda Rising. 实现领域驱动设计[M]. 滕云译. 北京：电子工业出版社，2014.
- [14] <https://martinfowler.com/articles/richardsonMaturityModel.html>.
- [15] Flavio Junqueira, Benjamin ReedZoo. Keeper：分布式过程协同技术详解[M]. 谢超译. 北京：机械工业出版社，2016.
- [16] [https://en.wikipedia.org/wiki/CAP\\_theorem](https://en.wikipedia.org/wiki/CAP_theorem).
- [17] <https://dl.acm.org/citation.cfm?doid=1394127.1394128>.
- [18] <https://martinfowler.com/bliki/CircuitBreaker.html>.
- [19] 张开涛. 亿级流量网站架构核心技术[M]. 北京：电子工业出版社，2017.
- [20] <https://martinfowler.com/articles/mocksArentStubs.html>.



[21] Rafal Kuc, Marek Rogozinski. Elasticsearch 服务器开发(第2版)[M]. 蔡建斌译. 北京:人民邮电出版社, 2015.

[22] <https://www.elastic.co/guide/en/logstash/current/index.html>.

[23] <https://tools.ietf.org/html/rfc6749>.

[24] <https://tools.ietf.org/html/rfc7519>.

[25] Gregor Hohpe, Bobby Woolf. 企业集成模式[M]. 荆涛, 王宇译. 北京: 中国电力出版社, 2006.

[26] <http://www.infoq.com/cn/presentations/team-building-implementation-in-distributed-world>.

[27] Project Management Institute. 项目管理知识体系指南(第5版)[M]. 许江林等译. 北京: 电子工业出版社, 2013.

[28] [https://en.wikipedia.org/wiki/Tuckman%27s\\_stages\\_of\\_group\\_development](https://en.wikipedia.org/wiki/Tuckman%27s_stages_of_group_development).

[29] Esther Derby, Esther Derby. 敏捷回顾: 团队从优秀到卓越之道[M]. 周全, 冯左鸣, 拓志祥, 李丽森译. 北京: 电子工业出版社, 2012.



非卖品!! 严禁(售卖和上传互联网平台)!!  
仅供对书籍质量进行鉴定甄别! 为是否购买正版实体书提供依据!!

# 微服务

## 设计原理与架构

MICROSERVICES  
Design Principle and Architecture

### 作者简介

郑天民 网名天涯兰, 日本足利工业大学信息工程学硕士, 研究方向为人工智能在大规模调度系统中的应用, 在SCI、EI等国际三大索引上发表学术论文4篇, 被引用达到50余次。10年软件行业从业经验, 在医疗、安防和电商行业都有所涉及, 主持和参与过多个大型企业级应用和移动互联网系统的开发和管理的工作, 前后担任系统分析架构师、部门经理、技术总监等职务。目前就职于一家业界领先的移动医疗互联网公司, 负责产品研发与技术团队管理工作。主持过十余个面向研发人员的技术和管理类培训课程, 善于提炼和抽象核心内容作为教学内容, 善于知识分享和技术人员培养, 对架构设计和技术管理有丰富的经验和深入的理解。著有《系统架构设计: 程序员向架构师转型之路》、《向技术管理者转型: 软件开发人员跨越行业、技术、管理的转型思维与实践》等书籍。

### 内容简介

本书内容主要包含实施微服务架构的一些方法论和工程实践, 首先, 通过对微服务架构的基本概念、服务建模、服务拆分和集成的介绍, 帮助读者全面理解微服务架构中的设计理念, 然后从微服务架构的基础组件、关键要素、实现框架以及管理体系等维度出发, 阐述实现微服务架构的工具和实践。最后, 本书还给出了从现有系统向微服务架构转型的思路、过程和案例分析。

本书面向立志于成为微服务架构师的后端服务开发人员, 读者不需要有很深的技术水平, 也不限于特定的开发语言; 不过, 熟悉 Java EE 常见技术并掌握一定系统设计基本概念, 有助于更好地理解书中的内容。同时, 本书也可以供具备不同技术体系的架构师同行参考阅读, 希望能给日常研发和管理工作带来启发和帮助。

免/费/提/供  
PPT等教学相关资料

 人邮教育  
www.rjiaoyu.com

教材服务热线: 010-81055256

反馈/投稿/推荐信箱: 315@ptpress.com.cn

人民邮电出版社教育服务与资源下载社区: www.rjiaoyu.com

ISBN 978-7-115-47882-5



9 787115 478825 >

ISBN 978-7-115-47882-5

定价: 59.80 元

封面设计: 董志桢